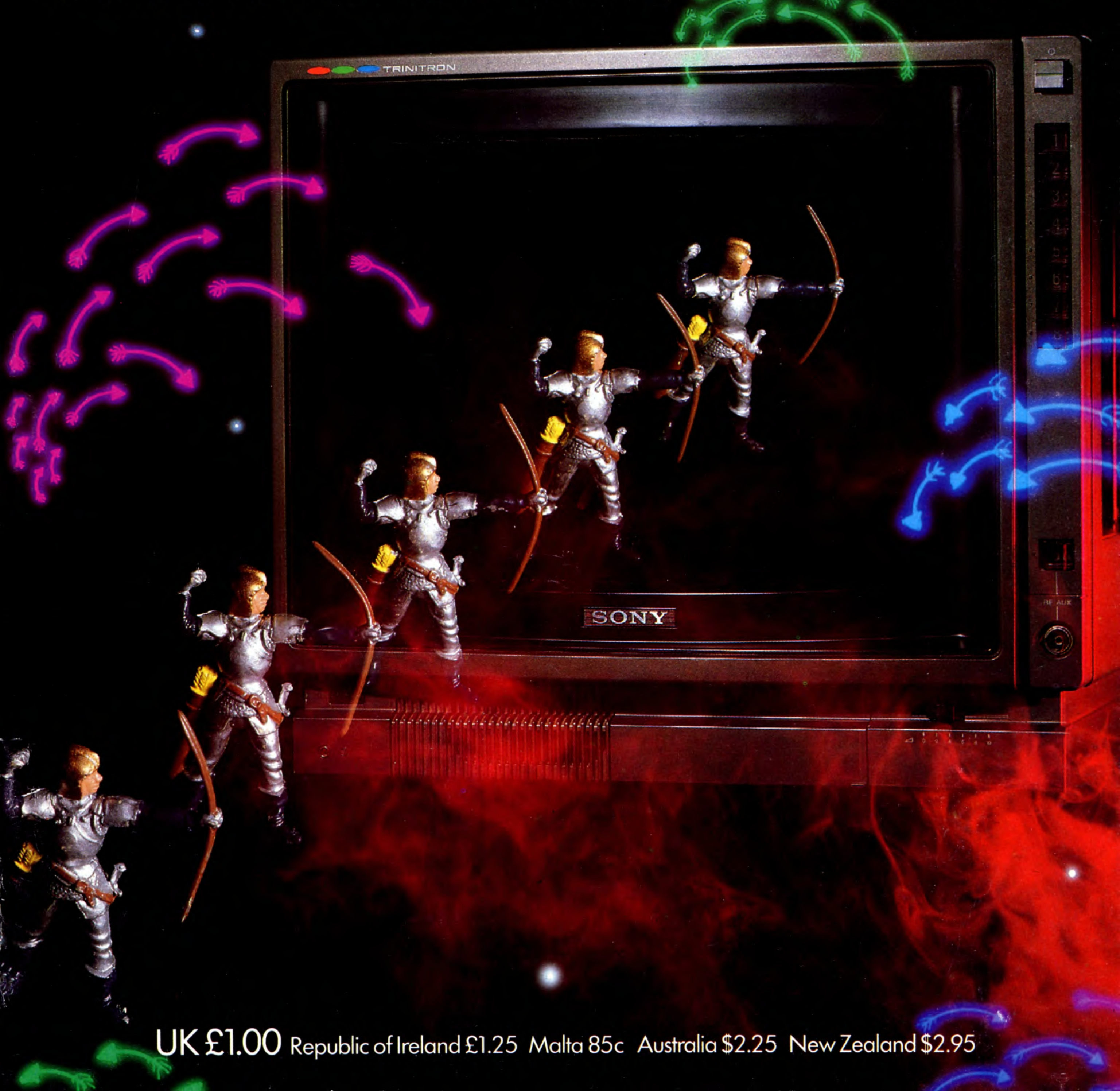
A MARSHALL CAVENDISH 43 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol. 4 **No 43**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS

Front cover, Graeme Harris. Page 1333, George Logan/Kuo Kang Chen. Page 1334, George Logan. Page 1336, George Logan/Spectrum. Pages 1338, 134Ø, 1342, 1344, Phil Dobson. Pages 1347, 1348, Graeme Harris. Pages 1352, 1353, 1355, Gary Wing. Page 1357, George Logan. Pages 1359, 1362, Jeremy Gower.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Cooper Clegg Web Offset Ltd, Gloucester and Howard Hunt Litho, London.

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

MUSIC COMPOSER PROGRAM

- WHAT IS A MUSIC COMPOSER
- HOW TO ENTER THE NOTES
- PLAYING A TUNE ON THE KEYBOARD
- CREATING A PIECE OF MUSIC

With this handy package, your computer becomes a notebook for your musical ideas—and a band to play back the masterpieces which you have just created

There are two ways of getting your computer to play a tune, either by encoding the tune in numbers the computer understands and writing a program to play it, or by using a music composer program that takes care of all the programming for you. All you have to do is enter the music.

Several articles in *INPUT* have shown you how to program your computer to play a piece of music. This is not too difficult but you do need a good knowledge of the BASIC sound commands—which often do not bear much relationship to music. Usually these require several numbers to define each note and you need to remember—or look up in the manual—the correct values for pitch, amplitude, duration and so on. Programming the Dragon and Tandy to play music is easier than the other computers but even with these you still have to deal with all the other details of programming.

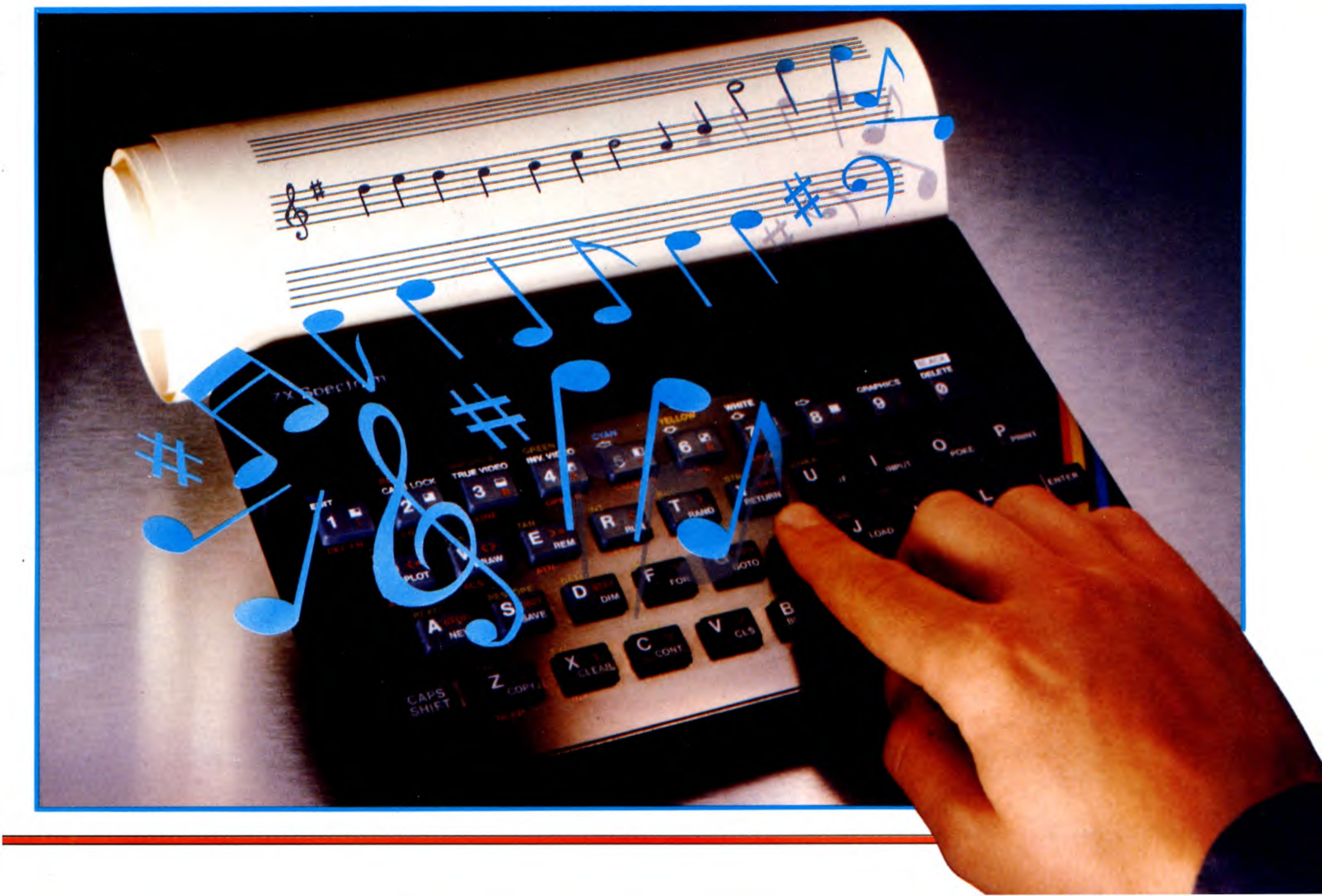
Another problem with writing a program to play music is that it is very difficult to alter or edit the tune later on and you can't hear how it sounds until the whole tune has been entered. Also, the music you've created is tied to the program and you need to write a new program for each new piece of music.

The music composer takes care of all these things for you leaving you free to concentrate on the music itself. And you don't need to know anything about programming in order to use it.

Most of the programs are quite long so they have been split into three parts. Enter the first part now and SAVE it ready to add the second part next time.

ENTERING THE NOTES

Once you've loaded the complete music composer into your computer you can enter a tune in a variety of ways. The exact method is described below for each computer, and there will be more detailed instructions with the last part of the program. But whichever method is used, you can alter the tune by changing, inserting or deleting notes until it sounds just right. And you can replay it at any time to



hear how you are getting on. You can also alter the tempo of the tune and change the octave. The Acorn and Commodore 64 programs also let you alter the sound of the note so you can even choose which 'instrument' plays the tune.

The Spectrum and Commodore 64, Dragon and Tandy programs all work in a similar way. You can choose whether you enter the notes by typing in their name such as A or F (the Spectrum and Commodore use a simple code) or you can choose to play the keyboard like a piano, with the computer automatically remembering the notes. You can mix the two methods if you like, playing parts of the tune and entering other parts one note at a time.

The Vic program uses the Super Expander cartridge which has many of the music composer routines already available. The tune is entered by typing in the names of the notes. You cannot enter notes by playing the keyboard, but you can hear them as you type in their names.

The Acorn program is quite different and is designed to make use of the high resolution graphics available on the BBC and Electron. Instead of entering the names of the notes, you are presented with musical staves and a selection of notes, rests and clefs which you can position on the staves using a few simple keypresses. You can very quickly produce a professional-looking musical score that you can replay using various instruments. It is also very easy to edit. If you insert or delete any notes the remaining notes automatically move to make room or close up the space.

Full instructions on the facilities will be given as the program develops.

```
10 BORDER Ø: PAPER Ø: INK 7: CLS
40 LET maxnotes=15ØØ: LET ct=Ø: LET
   tempo=.1: POKE 236Ø9,128: POKE
   23658,Ø: DIM m(35)
95 FOR i=1 TO 35: READ m(i): NEXT i
1ØØ DIM t(maxnotes+1): GOTO 11Ø
1Ø5 CLS : PRINT "Please wait. Tune being
   cleared"
11Ø FOR i=1 TO maxnotes: LET t(i)=Ø:
   NEXT i
19Ø CLS
2ØØ PRINT INVERSE 1;AT Ø,9;" MAIN
   MENU "; INVERSE Ø; '''TAB 5;"[1] — Play
   on keyboard"''TAB 5;"[2] — Enter
   Notes"''TAB 5;"[3] — Replay Tune"
21Ø PRINT 'TAB 5;"[4] — Edit Tune"''TAB 5;
   "[5] — Clear Tune"''TAB 5;"[6] — Save
   Tune"''TAB 5; "[7] — Load Tune"
27Ø PRINT "TAB 3;"Enter Option — (Q) to
   Exit";
3ØØ LET A$=INKEY$: IF A$="" THEN
   GOTO 3ØØ
31Ø LET A=CODE (A$)
32Ø IF (A<49 OR A>55) AND A<>113
   AND A<>81 THEN GOTO 19Ø
33Ø IF A=49 THEN GOSUB 1ØØØ
34Ø IF A=5Ø THEN GOSUB 2ØØØ
35Ø IF A=51 THEN GOSUB 3ØØØ
36Ø IF A=52 THEN GOSUB 4ØØØ
365 IF A=53 THEN GOTO 1Ø5
366 IF A=54 THEN GOSUB 5ØØØ
367 IF A=55 THEN GOSUB 6ØØØ
37Ø IF A=113 OF A=81 THEN STOP
38Ø GOTO 19Ø
1ØØØ CLS
1ØØ1 LET len=1: LET l$="Semi quaver"
1ØØ2 INPUT "[E]XTEND LAST TUNE OR
   [S]TART□ □ □NEW TUNE ?";i$
1ØØ3 IF i$<>"e" AND i$<>"s" THEN
   GOTO 1ØØ2
1ØØ4 LET num=1: IF i$="e" THEN LET
   num=ct*2+1: LET tn=num
1ØØ5 IF i$="s" THEN LET ct=Ø
1Ø1Ø PRINT "Play Notes on the Keys
   from"'"<Q> — lowest note
   to"'"<M> — highest note"'"Two and a
   half Octaves are"'"available, middle C being
   key I"''''
1Ø6Ø PRINT AT 17,Ø;"Note length="
1Ø7Ø FOR i=1 TO 1ØØ: NEXT i
1Ø75 LET O$=""
1Ø8Ø LET N$=INKEY$: IF N$="" THEN
   GOTO 1Ø75
1Ø85 IF N$=O$ THEN GOTO 1Ø8Ø
1Ø9Ø LET N=CODE (N$)
1Ø91 IF N=33 THEN LET len=1: LET
   l$="Semi quaver"
1Ø92 IF N=64 THEN LET len=2: LET
   l$="Quaver"
1Ø93 IF N=35 THEN LET len=4: LET
   l$="Crotchet"
1Ø94 IF N=36 THEN LET len=8: LET
   l$="Minim"
1Ø96 IF N=37 THEN LET len=16: LET
   l$="Semi-breve"
1Ø97 PRINT AT 17,13;"□ □";l$;
   "□□□□□□"
1Ø99 IF N<>32 AND N<>13 THEN GOTO
   11Ø9
11ØØ IF i$="s" THEN LET CT=INT
   (num/2): RETURN
11Ø2 LET CT=CT+INT ((num−tn)/2):
   RETURN
11Ø9 IF N<6Ø THEN LET index=N−47
111Ø IF N>9Ø THEN LET index=N−87
1112 IF N>=6Ø AND N<=9Ø THEN GOTO
   1Ø8Ø
1115 IF index=5 OR index=9 OR index=16
   OR index=2Ø OR index=21 THEN GOTO
   1Ø8Ø
1116 IF index=2 THEN LET t(num)=len: LET
   t(num+1)=−4: GOTO 1127
1118 IF index<=Ø THEN GOTO 1Ø8Ø
112Ø BEEP len/1Ø,m(index)
1125 LET t(num)=len: LET
   t(num+1)=m(index)
1126 PRINT AT 15,Ø;INT (num/2)+1;
   " Notes in store"
1127 LET num=num+2
113Ø LET O$=N$
114Ø GOTO 1Ø8Ø
```

```
1Ø POKE 5328Ø,11:POKE 53281,Ø
2Ø PRINT "♡▣▣▣▣▣▣▣▣▣
   ▣"TAB(11)"⊞C▟O▟M▟
   ▟S▟Y▟N▟T▟H"
3Ø PRINT TAB(11)"◆□□□□□□□
   □□□□□□□□□"
4Ø NS=Ø:TEMPO=9Ø:SID=54272
1ØØ DIM WF(2),AT(2),DE(2),SU(2),RE(2),
   CR(2),VOICE(2,5ØØ)
11Ø DIM PW(2),FR(2),SY(2),RM(2),FI(2),
```

```
   NS(2),H(96),L(96)
120 FOR I=1 TO 96:READ H:
   H(I)=H:NEXT I
130 FOR I=1 TO 96:READ L:
   L(I)=L:NEXT I
140 GOSUB 2005:GOSUB 9000:
   GOSUB 6000:GOSUB 2050
200 PRINT "♥▣♠"TAB(9)"COMSYNTH
   OPTIONS MENU"
220 PRINT"▣✝0 — PLAY ON
   KEYBOARD/REPLAY TUNE"
230 PRINT"▣1 — CHANGE VOICE 1
   PARAMETERS"
240 PRINT"▣2 — CHANGE VOICE 2
   PARAMETERS"
250 PRINT"▣3 — CHANGE VOICE 3
   PARAMETERS"
260 PRINT"▣4 — CHANGE GENERAL
   PARAMETERS"
270 PRINT"▣5 — ENTER NOTES"
280 PRINT"▣6 — EDIT TUNE"
305 PRINT"▣7 — INITIALISE PARAMETERS
   OF ALL VOICES"
310 PRINT"▣8 — SAVE TUNE"
315 PRINT"▣9 — LOAD TUNE"
320 PRINT"▣◘◘◘◘◘◘▒ENTER
   OPTION◘◘ — ◘◘(Q) TO EXIT◘
   ◘—◣"
325 R=0
326 LINE=3
330 GET A$:IF A$="" THEN 330
335 IF A$="Q" THEN SYS 58648:END
340 A=ASC(A$)—ASC("0")
345 IF A<0 OR A>9 THEN 330
350 ON A+1 GOSUB 2500,3000,3000,
   3000,4000,7000,7500,2000,9300,9400
360 GOTO 200
1000 PRINT "♥OPTION□";A;"□
   SELECTED"
1010 RETURN
2000 PRINT "♥"
2005 PRINT "▦▣"TAB(12)">◘
   INITIALISE◘<":FOR I=0 TO 2
2010 WF(I)=0:AT(I)=0:DE(I)=0:
```

```
   SU(I)=15
2020 RE(I)=0:PW(I)=2048:FR(I)=4291:
   SY(I)=0
2030 RM(I)=0:FI(I)=0
2040 NEXT I
2045 FF=1500: FR=0: VOL=15: LP=0:
   BP=0: HP=0: V3=0
2048 GOSUB 6000:RETURN
2050 FOR I=0 TO 2
2060 FOR J=0 TO 500
2070 VOICE(I,J)=0
2080 NEXT J,I
2090 RETURN
2500 PRINT "♥PRESS◘<P>◘TO PLAY
   ◘+◘[SHIFT KEY]◘TO STORE◘
   <R>◘TO REPLAY◘—"
2510 GET D$:IF D$="" THEN 2510
2520 IF D$<>"R" AND D$<>"P" AND
   D$<>"□" THEN RETURN
2521 IF D$="R" THEN 2900
2522 IF D$="P" THEN 2533
2523 INPUT "♥◘VOICE◘";VO:INPUT
   "OCTAVE◘";OC
2533 PRINT "♥PLAY ON TOP 2 ROWS OF
   KEYBOARD"
2534 PRINT"▣VOICE 1 ONLY SOUNDS"
2538 PRINT"▣▣<RETURN> TO FINISH"
2540 N=PEEK(203):GET Z$:IF N=64 THEN
   2560
2550 GOTO 2570
2560 POKE SID+4,CR(0):GOTO 2540
2570 IF N=1 OR N=60 THEN RETURN
2580 NN=12:N2=0:IF N=62 THEN
   HF=8:LF=97:NN=0
2590 IF N=59 THEN HF=8:
   LF=255:NN=1
2600 IF N=9 THEN HF=9:
   LF=104:NN=2
2610 IF N=8 THEN HF=9:
   LF=247:NN=3
2620 IF N=14 THEN HF=10:
   LF=143:NN=4
2630 IF N=17 THEN HF=11:
   LF=48:NN=5
2640 IF N=16 THEN HF=11:
   LF=218:NN=6
2650 IF N=22 THEN HF=12:
   LF=143:NN=7
2660 IF N=19 THEN HF=13:
   LF=78:NN=8
2670 IF N=25 THEN HF=14:
   LF=24:NN=9
2680 IF N=24 THEN HF=14:
   LF=239:NN=10
2690 IF N=30 THEN HF=15:
   LF=210:NN=11
2700 IF N=33 THEN HF=16:
   LF=195:NN=0:N2=1
2710 IF N=32 THEN HF=17:
   LF=195:NN=1:N2=1
2720 IF N=38 THEN HF=18:
   LF = 209: NN = 2:N2 = 1
2730 IF N = 35 THEN HF = 19:
   LF = 239:NN = 3:N2 = 1
2740 IF N = 41 THEN HF = 21:
   LF = 31:NN = 4:N2 = 1
2750 IF N = 46 THEN HF = 22:
   LF = 96:NN = 5:N2 = 1
2760 IF N = 43 THEN HF = 23:
   LF = 181:NN = 6:N2 = 1
2770 IF N = 49 THEN HF = 25:
   LF = 30:NN = 7:N2 = 1
2780 IF N = 48 THEN HF = 26:
   LF = 156:NN = 8:N2 = 1
2790 IF N = 54 THEN HF = 28:
   LF = 49:NN = 9:N2 = 1
2800 IF N = 51 THEN HF = 29:
   LF = 223:NN = 10:N2 = 1
2803 IF NN < > 12 THEN 2810
2805 GOTO 2540
2810 POKE SID,LF:POKE SID + 1,HF:
   POKE SID + 4,CR(0) + 1
2811 IF D$ = "P" THEN 2540
2812 M = VAL(STR$(NN) + STR$(OC + N2)):
   V = VAL(STR$(M) + STR$(VO)) - M*10:
   V = V - 1
2814 VOICE(V,NS(V)) = M:NS = NS + 1:NS
   (V) = NS(V) + 1:WAIT 197,64
2820 GOTO 2540
2900 CT = 0
2904 IF VOICE(0,CT) < > 0 OR VOICE(1,
   CT) < > 0 OR VOICE(2,CT) < > 0
   THEN 2908
2905 POKE SID + 4,CR(0):POKE SID + 11,
   CR(1):POKE SID + 18,CR(2)
2906 RETURN
```

Enter and SAVE these two programs separately.

```
10 VDU23,255,1,3,2,6,4,4,2,2
20 VDU23,254,192,64,96,32,64,64,128,128
30 VDU23,253,2,3,2,6,10,18,39,42
40 VDU23,252,128,0,0,0,0,0,192,32
50 VDU23,251,74,82,74,74,66,34,18,15
60 VDU23,250,16,16,16,16,16,32,32,192
70 VDU23,249,2,2,2,18,18,8,0,0
80 VDU23,248,15,16,32,112,112,0,0,0
90 VDU23,247,192,33,16,16,8,9,16,16
100 VDU23,246,0,0,0,1,2,12,56,0
110 VDU23,245,32,64,128,0,0,0,0,0
120 VDU23,243,0,0,124,130,130,130,124,0
130 VDU23,242,0,0,124,254,254,254,124,0
140 VDU23,241,0,0,0,0,0,0,0,0
150 VDU23,240,0,124,124,0,0,0,0,0
160 VDU23,239,0,0,0,0,0,0,124,124
170 VDU23,238,0,0,0,0,0,0,0,0
180 VDU23,237,0,0,32,16,8,8,16,48
190 VDU23,236,16,8,56,32,16,8,0,0
200 VDU23,235,0,0,64,120,16,16,32,32
210 VDU23,234,64,64,128,0,0,0,0,0
220 VDU23,233,0,0,64,120,16,16,160,96
230 VDU23,232,64,64,128,0,0,0,0,0
240 VDU23,231,36,36,126,36,36,36,126,36
250 ENVELOPE1,1,0,0,0,1,1,1,100,-2,0,
   -1,120,70
260 ENVELOPE2,1,1,-1,0,2,2,1,100,-2,0,
   -1,120,70
270 ENVELOPE3,1,1,-1,0,4,4,1,100,-2,0,
   -1,120,70
280 ENVELOPE4,1,0,0,0,1,1,1,127,1,-1,
   -126,120,70
290 ENVELOPE5,1,0,0,0,1,1,1,127,1,-1,
   -127,120,100
300 CHAIN"COMP"
```

Call this next program "COMP".

```
10 *KEY0|MMODE6|M*
   FX4|M*FX12,0|M
20 MODE4
30 PROCInit
40 REPEAT
50 PROCM(0,13)
60 IFA% = 67PROCChangespeed
70 IFA% = 68PROCDelete
80 IFA% = 69PROCEnvelope
90 IFA% = 73ANDF3% = 0ANDNO% <
   LT% + 1PROCInsert
100 IFA% = 76PROCLoad
110 IFA% = 80PROCPlaytune
120 IFA% = 81ANDF3% = 1THEN
   PROCInsertexit
130 IFA% = 83PROCSave
140 IFA% = 84PROCTie(NO% - 1,NO%)
150 IFA% = 86PROCV
160 IFA% > 13ANDA% < 91GOTO50
170 PROCno:IFNO% = LT% + 1:UNTIL0
180 VDU5
190 IFI% < 7THENPROCEnter
200 IFA% > 13ANDA% < 91GOTO60
210 IFI% = 12ANDF%PROCDrawtrebleclef
   (SE%):S(0,NO%) = 256:S(1,NO%) = 0:
   NO% = NO% + 1:F% = -1
220 IFI% = 13ANDNOTF%PROCDrawbassclef
   (SE%):S(0,NO%) = 257:S(1,NO%) = 0:
   NO% = NO% + 1:F% = -1
230 IFI% = 12ORI% = 13THENPROCno:GOTO 50
240 IFI% > 6ANDI% < 12PROCDrawrest
   (I% - 7,SE% + 64):S(0,NO%) = 251 + I%:
   S(1,NO%) = I%
250 NO% = NO% + 1
260 IFX% < = 1100GOTO320
270 IFH% < > 2MOVE65,SE% - 60:PRINT;
   NO%
280 IFH% = 2PROCDisplay(E%(1),NO% - 1):
   H% = 1:IFX% > 100GOTO320
290 X% = 20:H% = H% + 1:SE% = ST%(H%)
300 IFF%PROCDrawbassclef(SE%)ELSE
   PROCDrawtrebleclef(SE%)
310 E%(H%) = NO%
320 PROCno:VDU5:UNTIL0
330 DATA 11,7,4,2,1
340 DATA 1,2,3,4,5,6,8,9,10,12,16,17,18,20,
   24,32
350 DATA -16,-12,0,6,16,23,32,39,48,55,
   64,71,80,93,97,108,112,124
360 DATA 5,13,21,25,33,41,49,53,61,69,73,81,
   89,97,101,109,117,121,129,137,145,149,
   157,165,169,177,185,193,197,205
370 DEFPROCRead
380 D$ = "":REPEAT
390 FORC% = 0TO100:NEXT:*FX15,1
400 D% = GET
410 IFD% = 13THEN470
420 IFD% < 48ORD% > 57AND(D% < 127OR
   LEN(D$) = 0):VDU7:GOTO390
430 IFD% = 127D$ = LEFT$(D$,LEN
   (D$) - 1):GOTO460
440 IFLEN(D$) = 3VDU7:GOTO390
450 D$ = D$ + CHR$(D%)
460 VDUD%
470 UNTILD% = 13
480 D% = VAL(D$)
490 ENDPROC
```

Tandy users will need to make one or two changes to this program and these will all be given with part three.

```
10 CLEAR 5000
20 MX=250:R1$="whqes":R2$="12345":
   R3$="abcdefgCDFGAp":NY$="Q2W3
   ER5T6Y7U19O0P@AZSXDCVGBHNMK,L.;
   /∧"+CHR$(10)+CHR$(8)+
   CHR$(9)+"□"+CHR$(21)+CHR$
   (93)+CHR$(13)+CHR$(12)
30 DIM N$(MX),C$(12)
40 FOR I=1 TO 5:READ LE$(I),
   L2$(I):NEXT
50 FOR I=1 TO 12:READC$(I):NEXT
60 DATA WHOLE,1,HALF,2,QUARTER,4,
   EIGHTH,8,SIXTEENTH,16
70 DATA c,C,d,D,e,f,F,g,G,a,A,b
80 FOR I=1 TO MX:N$(I)="*****":
   NEXT
90 NN=0:OC$="3":TE=8:LE$="w□":
   LE=1
100 CLS:PRINT@4,"MUSIC COMPOSER
   MAIN MENU"
110 PRINT@96,"1: LOAD MUSIC FROM
   TAPE"
120 PRINT"2: SAVE CURRENT MUSIC TO
   TAPE"
130 PRINT"3: PLAY ON THE KEYBOARD"
140 PRINT"4: ENTER NOTES"
150 PRINT"5: CHANGE OVERALL TEMPO"
160 PRINT"6: LIST/EDIT NOTES"
170 PRINT"7: PLAY TUNE IN MEMORY"
180 PRINT"8: GLOBAL OCTAVE CHANGE"
190 PRINT"9: EXIT PROGRAM"
200 PRINT@454,"ENTER OPTION NUMBER";
210 A$=INKEY$:IF A$<"1" OR A$>"9"
   THEN 210
220 OP=VAL(A$)
230 ON OP GOSUB 1900,2020,480,760,420,
   1360,1560,1720,250
240 GOTO 100
250 CLS:PRINT" ARE YOU SURE
   (Y/N) ?":POKE329,255
260 A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 260
270 IF A$="Y" THEN CLS:END ELSE100
280 PRINT@132,"NOTE□□□OCTAVE□□
   □□LENGTH"
290 RT=0
300 FOR L=160 TO 320 STEP 32:
   PRINT@L:NEXTL:PRINT@160,"";
310 FOR L=NN-5 TO NN
320 IF L<1 THEN 410
330 PRINTUSING "###□□";L;
340 A$=N$(L)
350 B$=LEFT$(A$,1):IF B$>="a" AND
   B$<="g" THEN C$=CHR$(ASC
   (B$)-32)+"□" ELSE C$=
   B$+"#"
360 IF B$="p" THEN C$="-□"
370 PRINT C$;"□□□□□□□";
   MID$(A$,2,1);"□□□□□□";
380 PRINT LE$(INSTR(R1$,MID$
   (A$,3,1)));
390 IF MID$(A$,4,1)="." THEN PRINT"."
   ELSE PRINT
400 IF RT<>0 THEN RETURN
410 NEXT L:RETURN
420 CLS:PRINT@7,"CHANGE TEMPO
   OPTION"
430 PRINT:PRINT"CURRENT TEMPO
   IS";TE
440 PRINT@128,"ENTER NEW TEMPO
   VALUE";:INPUT ST
450 IF ST=0 THEN RETURN
460 IF ST<0 OR ST>255 THEN 440
470 TE=ST:RETURN
480 CLS:POKE329,255:C=VAL(OC$):IF
   C>3 THEN C=3:OC$="3"
490 PRINT@7,"'ORGAN' PLAY MODE"
500 PRINT@64,"OCTAVE:";C;TAB(14);
   "LENGTH:";LE$(INSTR(R1$,LEFT$(LE$,
   1)));
510 IF MID$(LE$,2,1)="." THEN PRINT"."
   ELSE PRINT
520 PRINT@448,"UP/DN=OCTAVE□□□
   □□□LEFT/RT=LENGTHENTER=
   MENU□□□□□□□□CLEAR=
   DELETE";
530 GOSUB280
540 IF NN=MX THEN PRINT@416,
   "MAXIMUM NUMBER OF NOTES
   ENTERED!";:FORD=1TO1000:NEXT:
   RETURN
550 I$=INKEY$:IF I$="" THEN 550
560 P=INSTR(NY$,I$):IF P=0
   THEN550
570 IF P>36 THEN600
580 OC$=MID$(STR$(C+INT((P-1)/
   12)),2):I$=C$(P-12*INT((P-
   1)/12))
590 GOTO700
```

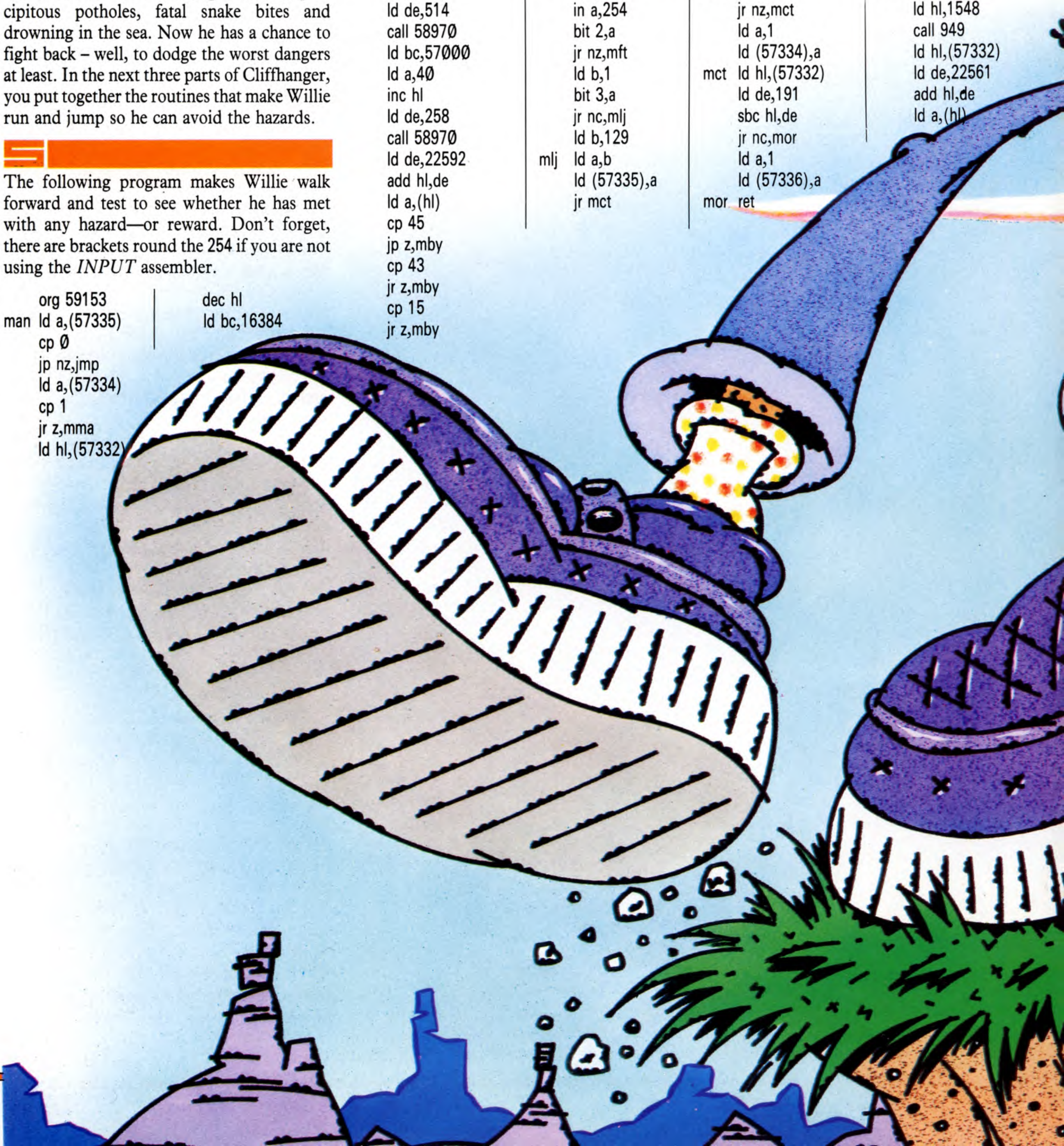
CLIFFHANGER: STEPPING OUT

Until now Willie has been a sitting target. It is time he started moving about. In part one of this three-part article, Willie takes his first faltering steps on the cliff

Willie has suffered crushing boulders, precipitous potholes, fatal snake bites and drowning in the sea. Now he has a chance to fight back – well, to dodge the worst dangers at least. In the next three parts of Cliffhanger, you put together the routines that make Willie run and jump so he can avoid the hazards.

The following program makes Willie walk forward and test to see whether he has met with any hazard—or reward. Don't forget, there are brackets round the 254 if you are not using the *INPUT* assembler.

```
     org 59153
man  ld a,(57335)
     cp Ø
     jp nz,jmp
     ld a,(57334)
     cp 1
     jr z,mma
     ld hl,(57332)
     dec hl
     ld bc,16384
     ld a,45
     ld de,514
     call 5897Ø
     ld bc,57ØØØ
     ld a,4Ø
     inc hl
     ld de,258
     call 5897Ø
     ld de,22592
     add hl,de
     ld a,(hl)
     cp 45
     jp z,mby
     cp 43
     jr z,mby
     cp 15
     jr z,mby
     ld a,Ø
     in a,254
     bit 2,a
     jr nz,mft
     ld b,1
     bit 3,a
     jr nc,mlj
     ld b,129
mlj  ld a,b
     ld (57335),a
     jr mct
mft  bit 3,a
     jr nz,mct
     ld a,1
     ld (57334),a
mct  ld hl,(57332)
     ld de,191
     sbc hl,de
     jr nc,mor
     ld a,1
     ld (57336),a
mor  ret
mma  ld de,3
     ld hl,1548
     call 949
     ld hl,(57332)
     ld de,22561
     add hl,de
     ld a,(hl)
```



- CHECKING BELOW
- SCANNING THE KEYBOARD
- WALKING FORWARD
- CHECKING FOR BOULDERS
- READY FOR NEXT CALL

The 'CLIFFHANGER' listings published in this magazine and subsequent parts bear absolutely no resemblance to, and are in no way associated with, the computer game called 'CLIFF HANGER' released for the Commodore 64 and published by New Generation Software Limited.

```
      cp 43
      jr z,mby
      cp 44
      jr z,mts
      cp 42
      jr z,mby
      ld de,32
      add hl,de
      ld a,(hl)
      cp 15
      jr z,mby
      cp 45
      jr z,mby
      cp 43
      jr z,mby
      ld hl,(57332)
      ld a,4Ø
      ld bc,57Ø16
      ld de,514
      call 5897Ø
      inc hl
      ld (57332),hl
mts   ld a,Ø
      ld (57334),a
      ret
mby   ld a,2
      ld (57336),a
      ret
jmp   ret
```

You can call this routine even though you don't have the rest of the man-moving routines in memory. Although this program calls them, it will not crash if you try to execute it before they are entered. A return, which will be overwritten later, is added at the end so that even if the missing routine is called, no damage will be done.

WALKABOUT

Memory location 57,335 contains the variable that tells Willie whether he is going to jump or not. The contents of this location are loaded up into the accumulator and compared to zero. If not zero, the man is going to jump and the processor jumps off the **jmp** routine. This is not in place at the moment and the processor simply jumps to a **ret** and returns to the place from which the routine was called.

If the man is not about to jump the processor continues. The contents of memory location 57,334 are then loaded into the accumulator. This location contains the so-called man-mode. There are two pictures of Willie—one with his legs together and one with his legs apart. When he is standing still, the picture with his legs together is printed over and over again. But when Willie is walking the two pictures are alternated. The man-mode tells the processor which picture to print up next.

The **cp 1** instruction looks to see which picture is to be printed up—Ø or 1. If picture 1 is required—the legs apart—**jr z,mma** sends the processor off to the routine that prints up that picture. And if picture Ø is required the processor continues.

THE INVISIBLE MAN

When Willie walks forward he opens his legs, then moves forward one character space, where he appears with his legs together. So an algorithm for this would run: print Willie with his legs closed, print Willie with his legs open and print him with his legs closed again one character square further forward. Willie with his legs open is two characters wide, and with his legs closed, he is only one character square wide. So this gives a fairly smooth movement.

But each time you print him with his legs closed you have to overprint him with sky in the square behind, otherwise you will get a trail of Willies with their legs apart all across the screen.

The instruction **ld hl,(57332)** loads Willie's position—which is stored in memory location 57,332—into the HL register. This is then decremented to move back one position.

BC is then loaded with 16,384. This is the address of the top of the screen—where there is sky. A is loaded with 45 and DE is loaded with 514. The block print routine at 58,97Ø is then called. Remember how this works. The contents of HL are the screen position. BC contains the data pointer. A specifies the colour—45 is cyan on cyan. And DE fixes the block size. 514 gives a block size of two by two character squares. D contains the number of rows that are in D and the number of columns that are in E—2 × 256 + 2 = 514.

So when the block routine is called it prints a block of two-by-two character square of cyan sky in the position immediately behind where you are about to print the new Willie. In other words, it blots out the old Willie who, with his legs apart, took up four character squares.

WILLIE REVISITED

Now you have to print up the new Willie with his legs together. The data for that starts at 57,ØØØ. So BC is loaded with 57,ØØØ. A is loaded with 4Ø, which is blue on cyan, Willie's colours.

HL is incremented to move it back again to Willie's new position—it was decremented earlier, remember. And DE is loaded with 258—giving a block of one by two (1 × 256 + 2 = 258), the size of Willie with his legs together. The block routine at 58,97Ø is then called again to print him up.

WHAT'S WILLIE WALKING ON?

As Willie has just moved forward one character square it would be a good idea to see what he is now standing on. If he has walked onto a snake, a pothole or is attempting to walk on water, you're going to have to tell Willie that he is dead.

The first thing to do is to check the attribute of the square under Willie's feet.

The attributes file starts at 22,528. The point presently in HL gives the screen position of Willie's head. But as you want the attribute of the square under his feet, which is two character squares down from there, you must add an extra 64.

So 22,592 is loaded into the DE register pair and added to HL. The result of the **add hl,de** instruction is always left in HL. The contents of the memory location now pointed to by HL are loaded into A. So the accumulator now contains the attribute of the character square immediately under Willie's feet.

This is then compared with 45, cyan on cyan, the attribute of sky which is what you have filled the pothole with. If 45 is found, the **jr z,mdy** instruction sends the processor off to a little routine which makes Willie die.

If what is under Willie's feet is not sky, the contents of the accumulator are compared to 43—magenta on cyan, the attribute for the snake—and 15—white on blue, the attribute for sea. If either of these is found the processor goes off to the **mby** routine again and kills Willie. If not, the processor goes sailing through, missing that routine, and Willie is saved.

WILL 'E JUMP?

Printing Willie on the screen one place forward with his legs together is effectively the end of the action. And provided he is not dead, now is the time to check whether Willie is about to move again. This is done by use of the **in** command (see page 731).

First, A is loaded with Ø. This means the whole keyboard is going to be scanned so that any block of keys can be used to control Willie's movement. M and N or J and K or U and I, or any of the other appropriate combinations can make Willie jump and walk, though on the instruction page, M and N were specified.

The **in** command is then used to look at port 254. And the instruction **bit 2,a** looks at bit two to check to see whether the M—or J or U—key has been pressed. All the bits are usually held high—that is, set to 1—but when they are pressed they are reset to Ø. So if the M key has been pressed, bit two is Ø.

In that case, the processor ignores the **jr nz** instruction. But if the key has not been pressed, the processor jumps on to the **mft** routine.

If the key has been pressed and Willie is required to jump, B is loaded with 1. Then bit three is tested in the same way.

If the N key is not pressed and the bit three is set to 1, the **jr nz** instruction makes the processor jump the next instruction. But if it is pressed, the **jr nz** instruction does not operate and B is loaded with 129.

So if the M key alone is pressed and Willie jumps but does not move forward, B contains 1. And if the M and N keys have been pressed and Willie jumps and moves forward, B contains 129.

Whichever route the processor takes, the contents of B are transferred into A and then loaded into memory location 57,335. This, you'll remember, is the memory location that was checked at the beginning of this routine to see whether Willie was going to jump or not.

The significance of these numbers 1 and 129 will be revealed in the next two parts of Cliffhanger which deal with jumping and jumping forward.

The **jr mct** instruction makes the processor jump over the next little routine.

FORWARD MARCH!

If M has not been pressed, the processor jumps to the **mft** routine. This checks to see whether the N key has been pressed on its own—in other words, if Willie is not jumping but simply marching forward.

Bit three is tested again to see if the N key has been pressed. Note that the input port 254 is still in the accumulator—the processor jumped straight here after bit two was tested.

If it hasn't been pressed, the **jr nz** instruction jumps straight on to the next routine. But if it has been pressed, A is loaded with 1 and this is stored in memory location 57,334. This is the so-called man-mode location which was checked at the beginning of the routine to see which picture of Willie was to be printed up on the screen. A 1 in this location means that Willie is to be printed with his legs open—in other words, he is about to walk forward.

REWARDING?

There is just one last thing to check—has Willie reached his just reward?

Willie's position is stored in memory location 57,332. And the contents of this location are loaded into HL. Then 191 is loaded into DE—191 is the screen location where the reward is printed.

The contents of DE are subtracted from the contents of HL. At this point, if Willie has not reached the reward the **jr nc** instruction jumps forward to the **ret** at the end of this part of the routine and returns. But if there is a carry, Willie has reached his reward and 1 is loaded into the accumulator and stored in 57,336. This is the memory location that is checked by the initialization routines and tell them to increment the score and start the next screen.

That done, the processor hits the **ret** and returns.

ONE PACE FORWARD

DE loaded with 3 and HL with 1548. Then the **BEEP** routine at 949 is called. This makes the walking sound effect. For the significance of the parameters 3 and 1548 see page 732.

But before Willie goes marching off it would be as well to check what is in front of him. His position is loaded into the HL register and 22,561 is added to it. Adding 22,561 points to the attribute of the character square 33 after the one pointed to by the contents of 52,332. That points to Willie's head, so 33 on is one square down—where his feet are—and one square forward—in other words, the square directly in front of his foot. This is then loaded into the accumulator by the **ld a,(hl)**.

The attribute of the square in front of Willie's foot is then compared to the attribute for the sea—15, white on blue; a snake—43, magenta on cyan; the slope—44, green on cyan; and the boulder—42, red on cyan. If the attribute for the sea, a snake or a boulder is found, the **jr z,mdy** instruction jumps the processor on to the routine that kills Willie. And if the attribute for the ground is found, the processor jumps on to a routine which keeps Willie stationary. Obviously, if there is ground in front of him, Willie cannot move until he is told to jump.

But checking the character square in front of Willie is not enough. You have to check the character square below the position he is about to move to as well. So DE is loaded with 32 and this is added to the contents of HL. This moves the pointer there 32 character squares along, so that it ends up one square below the place it started from.

The attribute of that character square is loaded into the accumulator by the **ld a,(hl)** instruction and the contents of the accumulator are compared to the attribute for the sea, a space and a snake again. If any of these are found, the **jr z,mdy** jumps on to the die routine.

Assuming Willie is still alive, HL is loaded with Willie's original position again. A is loaded with 4Ø—Willie's colour. BC is loaded with 57,Ø16—the address of the beginning of the data for the picture of Willie with his legs apart. And DE is loaded with 514—Willie with his legs apart takes up two by two character squares. Then the block printing routine at 58,87Ø is called which prints up on the screen the picture of Willie with his legs apart.

HL is then incremented and loaded back into memory location 57,332. This updates Willie's position so that next time this routine is called it starts from one place further to the right.

STAND STILL

The **mts** routine is called directly when Willie finds the slope in front of him and cannot move forward without jumping. But the processor also reaches it when it has finished printing up Willie with his legs apart.

A Ø is loaded into the accumulator and stored in memory location 57,334. This tells the processor to print up the picture of Willie with his legs together next time—either in the same place if Willie cannot move forward, or one space on if Willie with his legs apart has been printed on the screen and HL incremented.

The processor then returns.

DYING THE DEATH

If Willie has drowned in the sea, trodden on a snake, fallen down a pothole or been hit by a boulder, the **mdy** is called. This tells the rest of the program that Willie is dead.

It does this by loading 2 into the accumulator and storing it in 57,336. Elsewhere in the program there will be routine's checking this memory location.

After that has been done, the processor returns again.

Then there is another **ret**, with the label **jmp** alongside it. This is going to be overwritten next time, but it stops the routine crashing if **jmp** is called.

Before you start to move the man about you have to have a couple of other small routines in memory. Firstly, you have to check the keyboard or joystick to see which way Willie is supposed to move. Then you have to check to see what is underneath Willie's feet to see that he has not trodden on a snake, fallen down a pothole or stepped into the sea. Routines performing these simple tasks will be covered in this part of Cliffhanger.

The next part will provide routines that make Willie jump up and down and jump backwards and forwards. Then the whole lot will be combined in one routine which moves Willie whichever way you want him to go.

STICKS AND ZONES

The following routine looks at the keyboard and joystick and returns their values in location $Ø384.

```
        ORG 2Ø992
        LDA #Ø
        STA $Ø384
        LDA $DCØØ
        TAY
        AND #16
        BEQ JUMP
        LDA $Ø28D
        CMP #1
        BEQ JUMP
RETURN  TYA
        AND #8
        BEQ RUN
        LDA $ØØC5
        CMP #12
        BEQ RUN
        RTS
JUMP    LDA #2
        STA $Ø384
        JMP RETURN
RUN     LDA $Ø384
        ORA #1
        STA $Ø384
        RTS
```

A is loaded with Ø which is then stored into memory location $Ø384. This clears out any previous value from the storage area the routine is going to use to return the keyboard or joystick value.

Then A is loaded with the contents of $DCØØ. This is not a memory location, rather it is a register of the Complex Interface Adaptor (CIA). Data from port A appears in $DCØØ.

The contents of this register are transferred into Y to save them temporarily. Even though the TAY instruction means Transfer the contents of A into the Y register, again it is only a copying process. So the contents of $DCØØ are now in A—where they can be manipulated—and Y where they are stored.

The contents of the accumulator are then ANDed with 16. This isolates bit four which is the bit that carries the status of the joystick's fire button.

The bits of the joystick register are normally set to 1. But if that direction is chosen, or the fire button pressed, the appropriate bit is reset to Ø. So if the fire button has been pressed, the BEQ instruction branches the processor on to the JUMP routine.

If the fire button has not been pressed—or no joystick is attached—the processor moves on to the next instruction. It loads the accumulator with the contents of memory location $Ø28D which contains the flag which indicates whether the shift, CTRL or C= keys have been pressed.

The contents are compared with 1. If they are 1, the shift key has been pressed and the BEQ instruction sends the processor off to the JUMP routine.

FORWARD WITH WILLIE

The contents of the joystick status register stored in Y are then transferred back into the accumulator by the TYA. These are ANDed with 8 to check on bit three, which will be reset to zero if the joystick has been pushed to the right. If it has, the BEQ instruction will branch the processor to the RUN routine.

If the joystick has not been pushed to the right—or is not plugged in—the contents of memory location $ØØC5 are loaded into the accumulator. This returns the character string corresponding to the key being pressed.

The contents of the accumulator are then compared to 12, the keyboard code for Z. And if a Z has been pressed the BEQ instruction jumps to the RUN routine.

JUMP ABOUT

The JUMP routine loads the accumulator with 2 and stores it in $Ø384. Then it returns to check whether Willie is running too by jumping back to the label RETURN.

The RUN routine loads up the contents of $Ø384 and ORs them with 1. This sets bit zero, leaving bit one—which may or may not have been set earlier—alone. The result is stored back in $Ø384.

The processor then hits an RTS and returns.

DOWN BELOW

Before you start moving Willie, this little routine takes a look at the character immediately under him. It does this to check whether he is standing on firm ground or whether he's trodden on a snake, has fallen down a pothole or is standing in the sea.

```
ORG 21328        ROR A
LDA $CØ12        STA $Ø353
CLC              JSR $5ØØØ
ROR A            LDY #Ø
STA $Ø352        LDA ($FB),Y
LDA $CØ11        RTS
CLC
```

The double density coordinates for Willie that you worked out before are stored in memory locations $CØ11 and $CØ11. The double density X coordinate in $CØ12 is loaded into the accumulator.

To convert this back into a regular coordinate it has to be divided by two. This is done simply by rotating the contents of the accumulator one place to the left. Note that the carry flag is cleared first. A rotate moves whatever is in the carry flag into the empty bit of the register. So if, for any reason, the carry flag was set you'd get a spurious result here. Clearing the carry flag here precludes this. The result is stored in $Ø352 which is the memory location used to pass the X coordinate into the multiplication routine in part nine of Cliffhanger (see page 1146).

The next four instructions take the double density Y coordinates through the same process. Again the carry flag is cleared before the rotation—it would have been set by the previous rotation if a 1 had been shifted out of the least significant bit—and the least significant bit is ignored.

Note that the double density Y coordinate in $CØ12 is already pointing to the character square below Willie, so it does not have to be adjusted now.

The multiplication routine at $5ØØØ is then called. This, as you have seen, converts X and Y coordinates into a screen position which is stored in memory locations $FB and $FC.

The character displayed on the appropriate screen location is then loaded into the accumulator by LDA ($FB),Y. The Y is set to zero, but indexed indirect addressing is the only form of indirect addressing available. And this instruction takes as its base address the contents of memory locations $FB and $FC.

The processor then returns to the main man-moving routine, where it was called, with the character in the space below Willie in the accumulator.

The following routine prints up Willie on the screen and checks to see whether he is dead or not. Set up the computer as normal before you key it in.

```
30 FORPASS=ØTO3STEP3
40 RESTORE
90 DATA5,18,3,4
100 DATA 11,225,18,3
110 DATA 1,8,226,8
120 DATA10,227,18,3
130 DATA2,8,11,228
140 DATA8,10,229,4
150 DATAØ
160 FORA%=&1E67TO&1E7F:READ?A%:
   NEXT
210 DATA5,18,3,4
220 DATA11,225,18,3
230 DATA1,8,226,8
240 DATA10,227,18,3
250 DATA2,8,11,230
260 DATA8,10,231,4
270 DATAØ
280 FORA%=&1E80TO&1E98:READ?A%:
   NEXT
290 P%=&1E99
300 [OPTPASS
310 .PTman
320 LDX&7A
330 LDY&7B
340 JSR&1964
350 LDX#Ø
360 LDA&7C
370 AND#&1
380 BEQLb1
390 LDX#25
400 .Lb1
410 LDA&1E67,X
420 JSR&FFEE
430 INX
440 CMP#Ø
450 BNELb1
460 RTS
470 .Death
471 LDA&7A
472 AND#&1
473 BEQ Lb2
474 LDA&7B
475 AND#&1
476 BEQLb2
477 RTS
478 .Lb2
480 LDA#Ø
490 LDX&7A
500 LDY&7B
510 DEY
520 DEY
530 JSR&1DBD
540 CMP#13
550 BEQDead
560 CMP#14
570 BEQDead
580 INY
590 INY
591 LDA&70
592 AND#&F
600 JSR&1DBD
610 CMP#Ø
620 BEQDead
630 RTS
640 .Dead
650 LDA&7C
660 ORA#&4
670 STA&7C
675 JSR &1515
680 RTS
690 ]NEXT
```

You can only test the part of this program that prints Willie on the screen. To do so, load up all the other parts of Cliffhanger you have assembled so far and key in the following instructions:

```
CALL &1D77:CALL &1D9B:CALL &1E99
```

This should print Willie at his starting position.

A WALKING PICTURE

This program starts off with two blocks of DATA. The first block is in Lines 9Ø to 15Ø and is READ into a data table from &1E67 to &1E7F. It is the data for a picture of Willie standing still.

The second block is in Lines 21Ø to 27Ø and is READ into a data table from &1E8Ø to &1E98. This is the data for a picture of Willie walking.

MACHINE CODE MAN

Willie's position is stored in zero-page memory locations &7A and &7B. His X coordinate is in &7A and his Y coordinate is in &7B. So when you enter the machine code routine. The contents of &7A are loaded into the X register and the contents of &7B are loaded into the Y register.

The processor then jumps to the subroutine at &1964. This moves the graphics cursor to Willie's position. X is then loaded with Ø. This is being used to specify that Willie is in a whole square position. To get a smooth animation effect Willie is moved half a character square at a time. But he starts off in a whole character square position.

The contents of &7C are then loaded into the accumulator. This location carries the instructions for which way Willie is to move next. If bit seven is set it tells him to jump. If bit six is set it tells him to go to the left. If bit five is set it tells him to go to the right. If bit four is set it tells him to drop down one character square. If bit three is set it tells him to drop down two character squares. If bit two is set it tells him to drop down three character squares—he's dead. Bit one is not used and if bit zero is set Willie is in a half character square position.

The contents of &7C are then ANDed with 1. If the result is Ø—that is, bit zero is Ø—the

BEQ in Line 38Ø branches over the next instruction. But if the result is 1, the branch is not made and X is loaded with 25.

WHICH WILLIE?

The index in the X register is then used to load up the data for a picture of Willie. The 25, obviously, offsets the pointer from the data for Willie standing still and Willie walking.

So the LDA&1E67,X in Line 41Ø loads up the appropriate character of the picture data—either the walking or standing picture—and JSR&FFEE prints the character on the screen.

The INX in Line 43Ø then increments the offset to pick up the next piece of data. This is compared to Ø—you'll note that both blocks of data end with a Ø. So if a Ø is not found, the BNE instruction in Line 45Ø sends the processor back to the Lb1 label to pick up and print the next part of Willie on the screen.

But if a Ø is found, the BNE instruction in Line 45Ø does not operate and the processor hits the RTS in Line 46Ø and returns.

DEATH TO WILLIE!

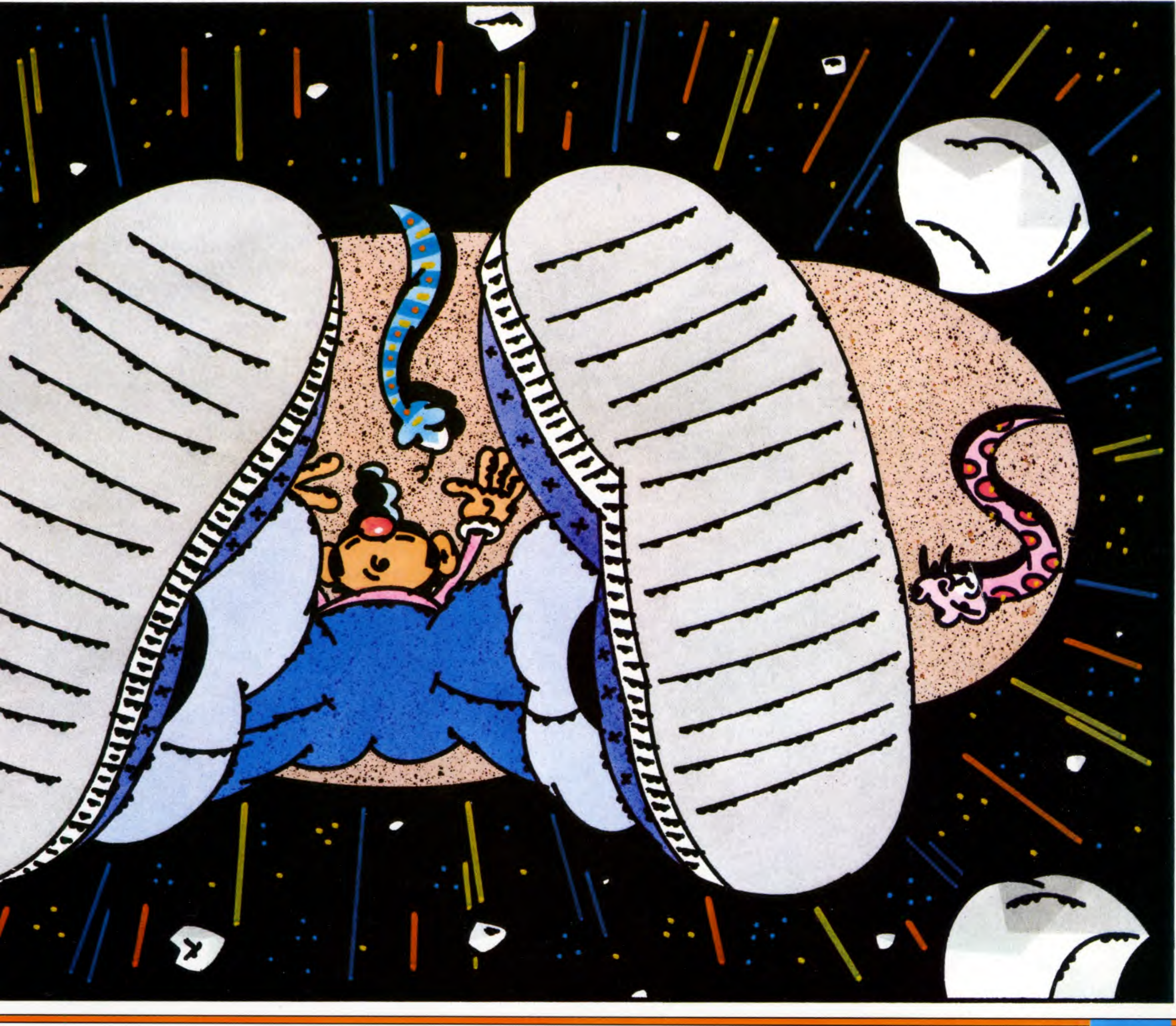
The next part of the routine checks whether Willie is dead or not and starts—appropriately enough—with the label Death in Line 47Ø. First the routine checks to see if Willie is in a half X or Y position—in that case he can't be dead and the processor returns.

First the X coordinate is loaded in the accumulator from zero-page memory location &7A. If the least significant bit is set, Willie is in half-character position. So it is ANDed with 1. This isolates bit zero—and if it is not set the processor branches forward. If it is set the processor continues and checks the Y coordinate.

If both are in half-character positions the processor hits the RTS in Line 477 and returns.

Then A is loaded with Ø and the X and Y registers are loaded with the X and Y coordinates of Willie from zero-page memory locations &7A and &7B. Y is then decremented twice so that it points to a position two character squares below Willie.

The processor then jumps to the routine at &1DBD (see page 1278) which returns the UDG number of the character in that square.



This is compared with 13 and 14, the UDG numbers for the snake's tongue and the snake's head. If either is found, Willie is dead and the BEQ instructions in Lines 54Ø and 56Ø branches forward to the Dead routine.

Y is then incremented twice to bring the Y pointer back to Willie's feet. A is loaded with contents of &7Ø which are ANDed with &F to isolate the background colour. And the routine at &1DBD is used again to return the UDG number of the character square there.

The number returned in A is compared to Ø. And if Ø is found, then the character at Willie's feet is not recognised. This means that the boulder UDG has got mixed up with the UDG for Willie's feet. The routine would recognise either of them separately, but when they are both printed in one character square, it cannot. In that case, the BEQ instruction in Line 62Ø sends the processor to the Dead routine.

If not, the processor hits the RTS in Line 83Ø and returns.

DEAD BUT NOT FORGOTTEN

If Willie is dead, the processor is directed to the Dead routine which starts in Line 64Ø. The contents of Willie's condition location, &7C, are loaded into the accumulator.

It is ORed with 4 which sets bit two. This is the bit that makes Willie drop down three character squares and buries him. The result is stored back in &7C where it can be referred to by other routines later. The processor jumps to the subroutine at &1515 which gives the death sound effect.

The processor then hits the RTS in Line 67Ø and returns.

The following program makes Willie walk forward and tests to see whether he has met with any hazard—or reward.

```
       ORG 19902
MAN    LDD 18249
       ANDB #31
       CMPB #30
       BNE MANI
       LDA #1
       STA 18252
MANI   LDA 18261
       LBNE JUM
       LDX 18249
       LEAX 544,X
       LDX ,X
       CMPX #$5555
       LBEQ MDY
       CMPX #$AAAA
       LBEQ MDY
       CMPX #$5FF5
       LBEQ MDY
       CLRB
       CLR 18264
       LDA #$BF
       STA $FF02
       LDA $FF00
       STA 18262
       LDA #$DF
       STA $FF02
       LDA $FF00
       STA 18263
       CMPA #$F7
       BNE MANA
       LDB #1
       LDA 18262
       CMPA #$F7
       BNE MAND
       LDB #129
MAND   STB 18261
       LDA 18264
       BNE MANC
       LDX 18249
       PSHS X
       BRA MMI
MANA   LDA 18262
       CMPA #$F7
       BNE MAND
       LDA #1
       STA 18264
       BRA MAND
MANC   LDX 18249
       LDU #1536
       JSR CHARPR
       LEAX 254,X
       JSR CHARPR
       LDX 18249
       LEAX 1,X
       PSHS X
       LEAX 353,X
       LDA ,X
       CMPA #$D5
       BEQ MDYA
       CMPA #$FF
       BEQ MDYA
       CMPA #$5Ø
       BEQ MTS
       LDA 18251
       BEQ MMO
MMI    LDX ,S
       LDU #17774
       JSR CHARPR
       LDX ,S
       LEAX 256,X
       JSR CHARPR
       CLR 18251
       BRA MANE
MMO    LDX, S
       LDU #17814
       JSR CHARPR
       LDX ,S
       LEAX 256,X
       LDU #17846
       JSR CHARPR
       LDA #1
       STA 18251
MANE   LDX ,S
       STX 18249
       PULS X
       RTS
MDYA   PULS X
MDY    LDA #2
       STA 18252
       RTS
MTS    PULS X
       LEAX -1,X
       PSHS X
       BRA MMI
JUM    RTS
CHARPR EQU 19402
```

This routine calls others which you don't have in memory yet, so do not call it unless you have put RTS's in the appropriate places to stop it crashing.

REWARDED

The first part of this routine looks to see whether Willie has reached his rewards. So Willie's screen position is loaded into the D register. As there are 32 positions across the screen—32 is 2↑5—you only need to look at the last five bits of the screen location to work out Willie's X coordinate. And if his X coordinate has reached 3Ø, the X coordinate of the reward, Willie must have reached it—as his Y position is fixed by the slope which he must be on or above.

So the contents of B are ANDed with 31—B is the low byte of the AB register pair that makes up D. This isolates the five least significant bits. These are then compared to 3Ø.

If the last five bits do contain 3Ø, 1 is loaded into A and stored in the die variable at 18,252. This tells the processor to bring on the next screen. If not, the BNE instruction branches the processor over these instructions.

JUMP OR HAZARD

A is loaded with the contents of 18,261, the jump variable. If they are not Ø, Willie is jumping and the processor makes the long branch to the JUM routine. It has to make a long branch because it is longer than 256 bytes.

Next you must check to see whether Willie has stepped on something nasty—a hole, a snake or the sea. So Willie's position is loaded up from 18,249 into the X register this time.

The LEAX 544,X then adds 544 to Willie's position. Willie is two character square's tall so to get down to his feet you need to add 2 × 256 which is 512. But there is one empty line of pixels between Willie's feet and the ground, so you add an extra 32 to clear; that makes 544.

The X register is then loaded with the contents of the two bytes under Willie's feet. It is compared to $5555, $AAAA and $5FF5. $5555 is the colour yellow, which means there is sky—that is, a hole—under Willie's feet. $AAAA is blue, or the sea. And $5FF5 is red on yellow, the colour of the snake. To understand how these colours are made up see pages 248 and 249.

If any of these is found under Willie's feet he is dead and the LBEQ instruction makes the processor branch to the MDY routine which kills Willie.

IN KEY

The next thing to do is to take a look at the keyboard to see if any key has been pressed. M and N control Willie's running and jumping so pressing them has to be detected.

Firstly, the B register has to be cleared. This is going to be used to carry the variable that controls whether Willie is going to jump. And memory location 18,264 is cleared as well because that is going to be used to store the variable that controls Willie's walking.

To look to see whether a key has been pressed you have to look at the keyboard matrix. Here you are trying to detect an M or an N. The letter M is in column four of row six and the letter N is in column four of row seven.

To examine the state of any key you have to write the row number into $FFØ2 and the status of that row comes back in $FFØØ. So to look at the N key you store $BF in $FFØ2. $BF is 1Ø111111 in binary—it's plain to see why this selects row seven. The result in $FFØØ is loaded into the accumulator and stored in 18,262.

M is examined in the same way. But this time $DF—11Ø11111 binary—is written into $FFØ2 and the result is stored in 18,263.

CMPA #$F7 then compares the result with F7, or 1111Ø111. You're looking to see whether the key in row four has been pressed—that is, grounded—remember.

If $F7 is not found—and Willie is not jumping—the BNE instruction branches the processor forward to MANA. But if it is found B is loaded with 1 to indicate a vertical jump.

The result of the examination of the row the N key is in, stored in 18,262, is then loaded into the accumulator and compared with $F7. Note that the N key is in the same column as the M key, though a different row. If it is not found, the BNE instruction branches over the next instruction. But if it is found, B is loaded with 129 to indicate that Willie is going to do a diagonal jump. Whatever is in B—1 or 129—is stored in 18,261.

WILLIE WALKER

The contents of memory location 18,264 are loaded into the accumulator. This is the location which stores the variable that indicates whether Willie is walking or not. It was cleared earlier in the program, remember, but it is not necessarily clear at this point because the routine which checks whether Willie is walking—which starts at the label MANA—branches back to MAND.

If the contents of memory location 18,264 are not zero and Willie is walking the BNE instruction branches the processor on to MANC where the program starts to make Willie walk. If not the X register is loaded with the contents of 18,249, the location which stores Willie's screen position, and pushes it onto the stack. BRA MMI then makes the processor branch to MMI where Willie is printed up on the screen in the position given by the last item on the stack.

If the M key is not pressed and Willie is not jumping, the processor jumps to MANA to see if Willie is walking. The result of the examination of the N key, stored in 18,262, is loaded back into the accumulator and compared to $F7. If it is not found, the BNE instruction branches the processor straight back to MAND. Memory location 18,264 is still clear so that Willie is printed up in the same place again. This means that he is, effectively, standing still.

But if the N key has been pressed and Willie is required to walk, 1 is loaded into the accumulator and stored in 18,264. When the processor now branches back to MAND, Willie will start to walk.

WILLIE WIPED OUT

When Willie starts to walk, the first thing you have to do is wipe him off the screen at his old position. Otherwise you would leave a trail of Willies across the screen.

X is loaded with Willie's screen position from 18,249 and U—the data pointer—is loaded with 1536, which is the top left-hand corner of the screen. So when the processor jumps to the CHARPR subroutine it prints two squares of sky over Willie. This gets rid of the top half of Willie.

To get rid of the bottom half of Willie, 254 is added to X and the CHARPR is called again.

Willie's position is loaded up into the X register again and 1 is added to it. This moves him forward one character square. It is pushed onto the stack to store it.

You now have to check that Willie is not going to walk into anything dangerous. LEAX 353,X increments the pointer to a byte directly in front of Willie's leg. And LDA ,X loads that byte up into the accumulator. This is then compared to $D5 which is the colour graphic for a snake's tongue and $FF, the boulder. If either of these are found Willie is about to die and the BEQ instruction takes the processor off to the MDYA routine which performs the slaughter.

It is also compared to $5Ø which is the colour graphic for the slope of the hill. And if this is found, the processor goes off to the MTS which prints Willie back in the same place and prevents him walking.

MOVING PICTURES

Memory location 18,251 is used as a flag to tell the processor which picture of Willie to print up next. There are two—one with his legs apart, the other with his legs together. When these are printed alternately as Willie is moved forward it gives the impression that he is walking.

The flag in 18,251 is loaded into the accumulator. If it is set to zero, the BEQ instruction branches the processor forward to the routine beginning with the label MMO. This prints up the picture of Willie with his legs apart. But if the contents of 18,251 are not zero and the flag is set to 1, the branch is not made and the processor continues into the MMI routine which prints up a picture of Willie with his legs together.

Note that if Willie is not walking the processor jumps straight in at MMI and prints up Willie with his legs together.

X is then loaded with the contents of the last item on the hardware stack. If you look back you will see that the last thing to be pushed onto the hardware stack is Willie's new position, one character square on from his last position.

U, the user stack pointer which is used as a data pointer by the CHARPR routine, is loaded with 17,774, the start address of the data for Willie with his legs together. The processor then jumps to CHARPR and prints up the top half of Willie.

The X register is loaded up again and incremented by 256. This moves the pointer down one character square, to the position of Willie's bottom half. CHARPR is then called again to print up Willie's bottom half. Note that U does not have to be loaded up again. It is the user stack pointer so it is automatically adjusted as the data is printed up on the screen.

Memory location 18,251 is then cleared so that the other picture of Willie—the one with his legs apart—is printed up next time. Then the processor branches to MANE.

Then following that is the MMO routine which prints up the other picture of Willie. It works in exactly the same way. U is set to 17,814 this time, though, which is the start address of the data for the picture of Willie with his legs apart. And at the end of the routine 1 is loaded into A and stored in 18,251. This is done to get Willie with his legs together printed up next time the man-moving routine is called.

Whichever Willie is printed the processor gets to the MANE routine. It loads Willie's new print position back up into X and stores it in 18,249. The last item on the stack is then pulled off into X to stop the hardware stack growing uncontrollably. The processor then returns.

DEAD OR IMMOBILE

If Willie has trodden on something nasty and is dead, the processor is sent to MDYA or MDY—which label it goes to depends on where he did it. If he stepped on something nasty at the beginning of the program, before Willie's new position was pushed onto the stack, it goes to MDY. But if Willie's new position has already been pushed onto the stack before Willie takes the fatal step, it goes to MDYA, so that the new position is pulled off the stack first.

A is then loaded with 2—the value that indicates that Willie is dead. And this is stored in 18,252, the location of the so-called die variable.

That done, the processor returns.

If Willie has walked into a slope, the processor is sent to MTS. Here Willie's new position is pulled off the stack again. This time though, the item pulled off the stack is used. It is decremented by 1 and pushed back onto the stack.

The processor then branches back to MMI and prints up Willie with his legs together in the same position as before so he effectively stands still.

WARGAMING: INTO BATTLE!

Here comes the crunch. At last combat can begin, with routines for missile combat and hand-to-hand fighting. Test morale on the beaten side and tot up the casualties

Cavendish Field is now almost completed; you can give your troops orders and move them about the battlefield. What is missing is the combat routines, and when you have entered these you will be able to try out your game.

What happens when the two sides engage in combat can be very complex, but again, decisions have to be made about what is going to go into your game. In the simplest case of combat, you could say something like: 'If two units meet, then the biggest one wins.' This assumes that the relative sizes of the units is the deciding factor in combat. Or you may decide that the outcome depends on the level of morale, or the number of surviving horses, or whatever. No-one really knows what wins battles, so in any wargame it is a matter of choice and program design to decide which factors are important.

MISSILES

In Cavendish Field there are two different kinds of combat—missile (arrows) and hand-to-hand. This first routine deals with missile combat.

```
1710 REM Missile Routine
1720 GOSUB 2540
1730 PRINT AT 18,Ø;"Unit□";
   sh;"□fires"
1740 LET fx=5: LET fy=5: LET gp=−1
1745 LET st=9
1750 IF sh>8 THEN LET st=1
1770 FOR m=st TO (st+7)
1780 LET tm=ABS (T(m,8)−T(sh,8)): LET
   ty=ABS (T(m,9)−T(sh,9))
1785 IF tm<fx AND T(m,1)<5 AND ty<fy
   THEN LET fx=tm: LET fy=ty: LET gp=m
1790 NEXT m
1800 IF gp=−1 THEN PRINT AT 19,Ø;
   "Nothing in range" : GOSUB 2410:
   RETURN
1810 LET C=8−T(gp,4)− ABS (fx−fy)
1820 IF gp<3 OR gp=9 OR gp=10 THEN
   LET C=C+1
1830 IF m(T(gp,8),T(gp,9))=2 THEN LET
   C=C−2
1840 IF T(gp,1)<>2 THEN LET C=C+1
1850 LET C=(C+(INT (T(sh,7)/4Ø))+ FN
   r(3))*1Ø
1860 LET T(gp,7)=T(gp,7)−C
1870 PRINT "That causes□";C;
   "□casualties on unit□";gp
1875 GOSUB 241Ø
1880 LET un=gp: GOSUB 22ØØ
1890 RETURN
```

```
1710 REM MISSILE ROUTINE
1720 GOSUB 254Ø
1730 PRINT"UNIT□";SH+1;"□FIRES"
1740 FX=5:FY=5:GP=−1
1745 BG=8
1750 IF SH>7 THEN BG=Ø
1770 FOR M= BG TO (BG+7)
1780 TM=ABS(T(M,7)−T(SH,7)):TY=ABS
   (T(M,8)−T(SH,8))
1785 IF TM<FX AND TY<FY AND
   T(M,Ø)<4 THEN FX=TM:FY=TY:GP=M
1790 NEXT M
1800 IF GP=−1 THEN PRINT "NOTHING
   IN RANGE":GOSUB241Ø:RETURN
1810 C=8−T(GP,3)−ABS(FX−FY)
1820 IF GP<2 OR GP=8 OR GP=9 THEN
   C=C+1
1830 IF M(T(GP,7),T(GP,8))=2 THEN
   C=C−2
1840 IF T(GP,Ø)<>2 THEN C=C+1
1850 C=(C+(INT(T(SH,6)/4Ø))+FNR(3))*
   1Ø
1860 T(GP,6)=T(GP,6)−C
1870 PRINT "UNIT□";GP+1;"□SUFFERS
   □";C;"□CASUALTIES"
1875 GOSUB 241Ø
1880 UN=GP:GOSUB 22ØØ
1890 RETURN
```

```
1710 DEF PROCfire(sht)
1720 PROCclean
1730 PRINT TAB(2,23);"Unit□";sht+1;"□
   fires"
1740 fx=5:fy=5:G%=−1
1750 IF sht>7 THEN st=Ø ELSE st=8
1770 FOR m=st TO (st+7)
1780 IF ABS(T%(m,7)−T%(sht,7))<fx□
   AND ABS(T%(m,8)−T%(sht,8))<fy□
   AND T%(m,Ø)<4 THEN fx=ABS
   (T%(m,7)−T%(sht,7)):fy=ABS(T%
   (m,8)−T%(sht,8)):G%=m
1790 NEXT
1800 IF G%=−1 THEN PRINT TAB(2,24);
   "Nothing in range":PROCpause:ENDPROC
1810 C%=8−T%(G%,3)−ABS(fx−fy)
1820 IF G%<2 OR G%=8 OR G%=9 THEN
   C%=C%+1
1830 IF FNmread(T%(G%,7),T%(G%,8))=2
   THEN C%=C%−2
1840 IF T%(G%,Ø)<> 2 THEN C%=C%+1
1850 C%=(C%+(T%(sht,6) DIV
   4Ø)+RND(3))*1Ø
1860 T%(G%,6)=T%(G%,6)−C%
1870 PRINT"That causes□";C%;
   "□casualties on unit□";
   G%+1:PROCpause
1880 PROCmorale(G%)
1890 ENDPROC
```

```
1710 REM MISSILE ROUTINE
1720 GOSUB 254Ø
1730 DRAW"BMØ,152":A$="UNIT"+
   STR$(SH)+"□FIRES":GOSUB 319Ø
1740 FX=5:FY=5:GP=−1
1745 ST=9
1750 IF SH>8 THEN ST=1
1770 FOR M=ST TO (ST+7)
1780 TM=ABS(T(M,8)−T(SH,8)):TY=ABS
   (T(M,9)−T(SH,9))
1785 IF TM<FX AND T(M,1)<5 AND
   TY<FY THEN FX=TM:
   FY=TY:GP=M
1790 NEXT M
1800 IF GP=−1 THEN DRAW"BMØ,16Ø":
   A$="NOTHING IN RANGE":GOSUB
   319Ø:GOSUB 241Ø:RETURN
1810 C=8−T(GP,4)−ABS(FX−FY)
1820 IF GP<3 OR GP=9 OR GP=1Ø THEN
   C=C+1
1830 IF M(T(GP,8),T(GP,9))=2 THEN
   C=C−2
1840 IF T(GP,1)<>2 THEN C=C+1
1850 C=(C+(INT(T(SH,7)/4Ø))+
   RND(3))*1Ø
1860 T(GP,7)=T(GP,7)−C
1870 DRAW"BMØ,16Ø":A$="CAUSES"+
   STR$(C)+"□CASUALTIES ON UNIT"+
   STR$(GP):GOSUB 319Ø
1875 GOSUB 241Ø
1880 UN=GP:GOSUB 22ØØ
1890 RETURN
```

- MISSILE COMBAT AND THE ARCHERS
- CHECKING RANGE
- HAND-TO-HAND COMBAT
- COUNTING CASUALTIES
- TESTING MORALE
- SCATTERING UNITS
- WINNERS AND LOSERS
- COMMENCING BATTLE
- INSTRUCTIONS

The missile combat routine is called every time the archers are ordered to open fire—in other wargames you might have more than one kind of unit which can open fire.

When a unit is ordered to fire G% (or gp or GP) is set to −1 in Line 174Ø. Next, the routine will decide if there is a suitable target. The coordinates of the shooting unit are compared with the coordinates of each of the enemy units. Line 178Ø compares the difference between the shooter's position and the target's. If the target is within a five-square range, G% (or gp or GP) is set to the target unit's number. If more than one target is within range, the nearest unit will be remembered.

If no target is found, G% (or gp or GP) will remain at −1, and the player will receive a NOTHING IN RANGE message. The routine then ends.

If the routine finds a target, the casualties are worked out. Casualties are stored in C% (or C).

Several factors affect whether losses are heavy or light. First, Line 181Ø starts with a value of 8 and subtracts the armour class of the target, and then a factor for range. Line 182Ø checks if the target is cavalry, and if it is, adds 1 to C% (or C). Line 183Ø checks if the target is in cover (terrain = type 2), then 2 is subtracted. Line 184Ø checks if the target is moving (recorded as the current order not being Halt) in which case 1 is added (archery attacks are more effective if the unit is charging). Finally, Line 185Ø adds the result to one fortieth of the strength of the shooting

unit, adds a small random number, and multiplies the result by ten. The result is the number of casualties suffered by the target unit. The casualties are subtracted from the current strength of the target unit, and a routine to test morale is called.

THE CLASH

Hand-to-hand combat is calculated in a similar way.

```
151Ø REM Combat
152Ø IF (us<9 AND th<9) OR (us>8 AND
   th>8) THEN RETURN
153Ø IF T(us,1)=5 OR T(th,1)=5 THEN
   RETURN
154Ø GOSUB 254Ø
155Ø PRINT AT 18,Ø;"Combat!!!"
156Ø LET at=INT ((T(us,7)−
   T(th,7))/5Ø)
157Ø LET at=at+T(us,3)−T(th,4)+T(us,
   5)+FN r(5)
158Ø IF ABS (T(us,2)−T(th,2))< >2 THEN
   LET at=at+2
159Ø IF us<3 OR us=9 OR us=1Ø THEN
   LET at=at+1
16ØØ LET dr=INT ((T(th,7)−T(us,7))/6Ø)
161Ø LET dr=dr+T(th,3)−T(us,4)−T(th,
   5)+m(T(th,8),T(th,9))+FN r(3)+2
1615 LET wn=th: LET lo=us
162Ø IF at>dr THEN LET wn=us: LET lo=th
163Ø LET wc=INT (T(wn,7)/1Ø): IF wc<1
   THEN LET wc=1
164Ø LET T(wn,7)=T(wn,7)−wc
165Ø LET lc=INT (T(lo,7)/5): IF lc<1 THEN
   LET lc=1
166Ø LET T(lo,7)=T(lo,7)−lc
167Ø PRINT wn;"□loses□";wc;"□";lo;"□
   loses□";lc
168Ø GOSUB 241Ø
169Ø LET un=lo: GOSUB 22ØØ
17ØØ RETURN
```

```
151Ø REM COMBAT
152Ø IF(US<8 AND TH<8)OR(US>7 AND
   TH>7)THEN RETURN
153Ø IF T(US,Ø)=4 OR T(TH,Ø)=4 THEN
   RETURN
154Ø GOSUB 254Ø
155Ø PRINT "COMBAT!!!"
156Ø AT=INT((T(US,6)−T(TH,6))/5Ø)
157Ø AT=AT+T(US,2)−T(TH,3)+
   T(US,4)+FNR(5)
158Ø IF ABS(T(US,1)−T(TH,1))< >2 THEN
   AT=AT+2
159Ø IF US<2 OR US=8 OR US=9 THEN
   AT=AT+1
16ØØ DF=INT((T(TH,6)−T(US,6))/6Ø)
161Ø DF=DF+T(TH,2)−T(US,3)+T(TH,
   4)+M(T(TH,7),T(TH,8))+FNR(3)+2
1615 WN=TH:LO=US
162Ø IF AT>DF THEN WN=US:LO=TH
163Ø WC=INT(T(WN,6)/1Ø):
   IF WC<1 THEN WC=1
164Ø T(WN,6)=T(WN,6)−WC
165Ø LC=INT(T(LO,6)/5):IF LC<1 THEN
   LC=1
166Ø T(LO,6)=T(LO,6)−LC
167Ø PRINT WN+1;"□LOSES□";WC;
   "□";LO+1;"□LOSES□";LC
168Ø GOSUB 241Ø
169Ø UN=LO: GOSUB 22ØØ
17ØØ RETURN
```

```
1510 DEF PROCcombat(us,th)
1520 IF (us<8 AND th<8) OR (us>7 AND
   th>7) THEN ENDPROC
1530 IF T%(us,Ø)=4 OR T%(th,Ø)=4 THEN
   ENDPROC
1540 PROCclean
1550 PRINT TAB(2,23);“Combat!!!”
1560 att=(T%(us,6)-T%(th,6)) DIV 5Ø
1570 att=att+T%(us,2)-T%(th,3)+T%(us,
   4)+RND(5)
1580 IF ABS(T%(us,1)-T%(th,1))< >2
   THEN att=att+2
1590 IF us<2 OR us=8 OR us=9 THEN
   att=att+1
16ØØ def=(T%(th,6)-T%(us,6)) DIV 6Ø
161Ø def=def+T%(th,2)-T%(us,3)+T%
   (th,4)+FNmread(T%(th,7),T%(th,8))+
   RND(3)+2
162Ø IF att>def THEN win=us:
   lose=th□ELSE win=th:lose=us
163Ø wc=T%(win,6) DIV 1Ø:IF wc<1 THEN
   wc=1
164Ø T%(win,6)=T%(win,6)-wc
165Ø lc=T%(lose,6) DIV 5:IF lc<1 THEN
   lc=1
166Ø T%(lose,6)=T%(lose,6)-lc
167Ø PRINT win+1;“□loses□”;wc;“□”
   lose+1;“□loses□”;lc
168Ø PROCpause:PROCmorale(lose)
17ØØ ENDPROC
```

```
151Ø REM COMBAT
152Ø IF (US<9 AND TH<9) OR (US>8
   AND TH>8) THEN RETURN
153Ø IF T(US,1)=5 OR T(TH,1)=5 THEN
   RETURN
154Ø GOSUB 254Ø
155Ø DRAW“BMØ,144”:A$=“COMBAT
   □□□”:GOSUB 319Ø
156Ø AT=INT((T(US,7)-T(TH,7))/5Ø)
157Ø AT=AT+T(US,3)-T(TH,4)+T
   (US,5)+RND(5)
158Ø IF ABS(T(US,2)-T(TH,2))< >2 THEN
   AT=AT+2
159Ø IF US<3 OR US=9 OR US=1Ø THEN
   AT=AT+1
16ØØ DF=INT((T(TH,7)-T(US,7))/6Ø)
161Ø DF=DF+T(TH,3)-T(US,4)+
   T(TH,5)+M(T(TH,8),T(TH,9))+RND
   (3)+2
1615 WN=TH:LO=US
162Ø IF AT>DF THEN WN=US:LO=TH
163Ø WC=INT(T(WN,7)/1Ø):IF WC<1 THEN
   WC=1
164Ø T(WN,7)=T(WN,7)-WC
165Ø LC=INT(T(LO,7)/5):IF LC<1 THEN
   LC=1
166Ø T(LO,7)=T(LO,7)-LC
167Ø DRAW“BMØ,16Ø”:A$=STR$(WN)+
   “□LOSES”+STR$(WC)+“□”+STR$
   (LO)+“□LOSES”+STR$(LC):
   GOSUB 319Ø
168Ø GOSUB 241Ø
169Ø UN=LO:GOSUB 22ØØ
17ØØ RETURN
```

The routine is called every time two units meet and RETURNs, or ENDPROCs on the Acorns, immediately if the two units belong to the same army. If the unit is being routed, Line 153Ø RETURNs also.

The main difference between hand-to-hand combat and missile combat is that both sides have something to say in the affray. This means that hand-to-hand combat is a messier business requiring two sets of casualties to be calculated.

In Line 156Ø the attackers start with one fiftieth of the difference between the two sides. Next, the difference between the attacker's weapon and the defender's armour is added, along with the value of the attacker's morale, and a random number up to five.

The attackers also gain a bonus if the enemy is not directly facing them. This will be the case either if the unit is not moving directly towards the attacker, or (when the unit has halted) if it was not moving in that direction last time it moved. This is why the 'direction' element of the troop array is never reset, and must always hold a direction. Historically, it seems that attacking the rear or flank was one of the most significant of combat factors. Finally, the attackers get a bonus of one point if they are cavalry.

Defenders are treated similarly. Starting at Line 16ØØ, they lose one sixtieth of the difference in numbers, plus the difference between the defender's weapon class and the attacker's armour class, plus the defender's morale, plus a bonus for the terrain they are defending, plus a fixed bonus of 2 and a random factor of up to 3. The random factor is intended to represent the fact that defending is easier than attacking, but sometimes the bloodlust of the attackers can intervene to overpower the innate advantage of defending.

The attackers will now have a value (at), as will the defenders (df). The side with the largest value wins that round of that fight, and therefore, loses only one tenth of its strength in casualties. The lower's forces are reduced by one fifth. Finally, the morale test is called for the loser.

THE MORALE MINORITY

The psychological factor is very important in warfare. It is very unlikely that an army will win a battle, equipped with the most powerful tanks in the world, if the troops hate their generals, think that the enemy have a just cause, or don't like the idea of fighting anyway. There are thousands of factors involved in the individual psychology of soldiers, and no game has ever come close to representing them all. At the most ridiculous level, the attitude of a warrior to the current fight can depend on whether he had a good night's sleep, or not, or on the quality of his last meal.

Not only can morale be complicated, but it can effect just about every aspect of the battle, giving a boost, or hampering the sides. In Cavendish Field, morale is only tested when a unit loses a fight, or is fired upon.

```
22ØØ REM Unit Morale Test
221Ø IF T(un,6)-T(un,7)<((T(un,6)/1ØØ)*
   ((T(un,5)+2))) THEN RETURN
222Ø GOSUB 254Ø
223Ø PRINT AT 18,Ø;"Losses are too great."
224Ø PRINT AT 19,Ø;"Unit□";
   un;"□disintegrates"
225Ø GOSUB 241Ø
226Ø LET T(un,1)=5
227Ø PRINT AT T(un,8),T(un,9);"□"
228Ø RETURN
```

```
22ØØ REM UNIT MORALE TEST
221Ø IF T(UN,5)-T(UN,6)<((T(UN,5)/1ØØ)*
   ((T(UN,4)+2)*1Ø)) THEN RETURN
222Ø GOSUB 254Ø
223Ø PRINT "LOSSES ARE TOO GREAT"
224Ø PRINT "UNIT□";UN+1;"□
   DISINTEGRATES"
225Ø GOSUB 241Ø
226Ø T(UN,Ø)=4
227Ø P=T(UN,7):Q=T(UN,8):
   GH=32:GOSUB 26ØØ
228Ø RETURN
```

```
22ØØ DEF PROCmorale(un)
221Ø IF T%(un,5)-T%(un,6)<((T%(un,5)/
   1ØØ)*((T%(un,4)+2)*1Ø)) THEN
   ENDPROC
222Ø PROCclean
223Ø PRINT TAB(2,23);"Losses are too great."
224Ø PRINT TAB(2,24);"Unit□";un+1;"□
   disintegrates"
225Ø PROCpause
226Ø T%(un,Ø)=4
227Ø PRINT TAB(T%(un,7)+1,T%(un,8)+1);
   "□"
228Ø ENDPROC
```

```
22ØØ REM UNIT MORALE TEST
221Ø IF T(UN,6)-T(UN,7)<((T(UN,6)/1ØØ)*
   ((T(UN,5)+2)*1Ø)) THEN RETURN
222Ø GOSUB 254Ø
223Ø DRAW"BMØ,152":A$="LOSSES ARE
   TOO GREAT":GOSUB 319Ø
224Ø DRAW"BMØ,16Ø":A$="UNIT"+STR$
   (UN)+"□DISINTEGRATES":GOSUB 319Ø
225Ø GOSUB 241Ø
226Ø T(UN,1)=5
227Ø X9=T(UN,9)*8:Y9=T(UN,8)*8:LINE
   (X9,Y9)-(X9+7,Y9+7),PRESET,BF
228Ø RETURN
```

Line 221Ø subtracts the current strength from the initial strength and compares it with the base morale of the unit. If the unit has suffered 30% casualties and is cowardly, it disintegrates and takes no further part in the battle. On the other hand, a unit which has a higher morale can suffer up to 70% casualties before disintegrating.

If you feel the routine is too clear-cut, you may want to extend it by adding a small random factor, considering if the unit was in cover, or examining the overall situation (how many enemy units are left, how many friendly units are nearby?).

When a unit fails the morale test, a message to that effect is displayed by Lines 223Ø and 224Ø. The command element of the array is set to five (or four in the case of the Acorn and Commodore), which means that the unit has been routed. The unit is blanked from the screen, and ignored in all further requests for orders and combat. However, because there are deserters milling all over the battlefield, the unit can still be an obstacle to movement, all the more hazardous because it is 'invisible' to the commander.

THE ARMISTICE

There are a few finishing touches to be added to the program: first, a victory condition.

```
229Ø REM Test for victory
23ØØ LET gd=Ø: LET bd=Ø
231Ø FOR m=1 TO 8
232Ø IF T(m,1)<>5 THEN LET gd=gd+1
233Ø IF T(m+8,1)<>5 THEN LET
   bd=bd+1
234Ø NEXT m
235Ø IF gd>bd*2 OR (bd<2 AND gd>2)
   THEN LET vc=1
236Ø IF bd>gd*2 OR (gd<2 AND bd>2)
   THEN LET de=1
237Ø RETURN
238Ø REM End Message
239Ø IF vc=1 THEN PRINT "VICTORY!!"
2395 IF de=1 THEN PRINT "A crushing
   defeat"
24ØØ RETURN
```

```
229Ø REM TEST FOR VICTORY
23ØØ GD=Ø:BD=Ø
231Ø FOR M=Ø TO7
232Ø IF T(M,Ø)<>4 THEN GD=GD+1
233Ø IF T(M+8,Ø)<>4 THEN BD=BD+1
234Ø NEXT M
235Ø IF GD>BD*2 OR (BD<2 AND GD>2)
   THEN VC=1
236Ø IF BD>GD*2 OR (GD<2 AND BD>2)
   THEN DE=1
237Ø RETURN
238Ø REM MESSAGE
239Ø IF VC=1 THEN PRINT "VICTORY!!"
2395 IF DE=1 THEN PRINT "A CRUSHING
   DEFEAT"
24ØØ RETURN
```

```
229Ø DEF PROCvicdef
23ØØ gd=Ø:bd=Ø
231Ø FOR m=ØTO7
232Ø IF T%(m,Ø)<>4 THENgd=gd+1
233Ø IF T%(m+8,Ø)<>4 THEN bd=bd+1
234Ø NEXT
235Ø IF gd>bd*2 OR (bd<2 AND gd>2)
   THEN vic=TRUE
236Ø IF bd>gd*2 OR (gd<2 AND bd>2)
   THEN def=TRUE
237Ø ENDPROC
238Ø DEF PROCmess
239Ø IF vic=TRUE THEN PRINT "VICTORY!"
   ELSE PRINT"A crushing defeat"
24ØØ ENDPROC
```

```
229Ø REM TEST FOR VICTORY
23ØØ GD=Ø:BD=Ø
231Ø FOR M=1 TO 8
232Ø IF T(M,1)<>5 THEN GD=GD+1
233Ø IF T(M+8,1)<>5 THEN BD=BD+1
234Ø NEXT M
235Ø IF GD>BD*2 OR (BD<2 AND GD>2)
   THEN VC=1
236Ø IF BD>GD*2 OR (GD<2 AND BD>2)
   THEN DE=1
237Ø RETURN
238Ø REM END MESSAGE
239Ø IF VC=1 THEN DRAW"BMØ,176":A$=
   "VICTORY":GOSUB 319Ø
2395 IF DE=1 THEN DRAW"BMØ,176":A$=
   "A CRUSHING DEFEAT":GOSUB 319Ø
24ØØ RETURN
```

The routine looks to see if one side has twice as many units as the other, or if one side has less than two units. A message is displayed to display the outcome.

Other victory conditions could be added. One factor which decided many medieval battles was the death of the leader, but adding this condition to the game could have its own problems. The game then simply becomes centred around one unit.

Another condition might be that half of one army had reached the opponent's baseline. Half could be measured either in units, or in total strengths of units.

Yet another criterion for victory could be the absolute number of casualties caused, in which case a variable containing the total number of casualties would have to be set up to count the dead after each exchange.

LET BATTLE COMMENCE

You now have all the routines that make up Cavendish Field. All you need now to marshal your troops is the main loop.

```
10 CLEAR
30 GOSUB 190
40 GOSUB 470
50 GOSUB 860
60 REM Dummy for Repeat loop
70 FOR i=1 TO 8
80 IF T(i,1)<4 THEN GOSUB 1380: IF y>2
   THEN GOSUB 1900
90 IF T(i,1)<5 THEN INK 1: PRINT AT
   T(i,8),T(i,9);u$(i)
100 NEXT i
110 FOR e=9 TO 16
120 IF T(e,1)<4 THEN GOSUB 2140
130 NEXT e
140 GOSUB 1020
150 GOSUB 2290
160 IF vc<>1 AND de<>1 THEN GOTO
   60
170 GOSUB 2380
180 STOP
2410 REM Delay
2420 PRINT AT 21,7;"{PRESS ANY KEY}"
2425 LET g$=INKEY$: IF g$="" THEN
   GOTO 2425
2430 RETURN
```

```
35 GOSUB 190
40 GOSUB 470
50 GOSUB 860
60 REM DUMMY FOR REPEAT LOOP
70 FOR I=Ø TO 7
75 Y=Ø
80 IF T(I,Ø)<>4 THEN GOSUB 1380:IF
   Y>2 THEN GOSUB 1900
90 IF T(I,Ø)<>4 THEN P=T(I,7):Q=T(I,8):
   GH=VAL(MID$(U$,I+1,1))+67:CL=1:
   GOSUB 2600
100 NEXT I
105 GOSUB 2540
110 FOR E=8 TO 15
120 IF T(E,Ø)<>4 THEN GOSUB 2140
130 NEXT E
140 GOSUB 1020
150 GOSUB 2290
160 IF VC<>1 AND DE<>1 THEN 60
170 GOSUB 2380
180 END
2410 REM DELAY
2420 PRINT "□□□□□□□□□
   [PRESS ANY KEY]"
2425 GET G$:IF G$=""THEN 2425
2430 RETURN
```

```
10 MODE1
30 PROCinit:PROCcr:PROCds
60 REPEAT
70 FORi=ØTO7
80 IF T%(i,Ø)<>4 THEN PROCunitsel : IF
   yn>2 THEN PROCactsel
90 COLOUR1:IF T%(i,Ø)<4 THEN PRINTTAB
   (T%(i,7)+1,T%(i,8)+1);MID$(unst$,
   i+1,1)
100 NEXT
110 FORen=8TO15
120 IF T%(en,Ø)<>4 THEN PROCensl
130 NEXT
140 PROCeffect:PROCvicdef
160 UNTIL vic OR def
170 PROCmess
180 END
2410 DEF PROCpause
2420 PRINT TAB(10,30);"[PRESS ANY
   KEY]":G=GET
2430 ENDPROC
```

```
10 CLEAR 5ØØ:PMODE 3,1:COLOR 2,1:PCLS:
   SCREEN1,Ø:DU=RND(-TIMER)
30 GOSUB 19Ø
40 GOSUB 47Ø:GOSUB 313Ø
50 GOSUB 86Ø
60 REM
70 FOR I=1 TO 8
80 IF T(I,1)<4 THEN GOSUB 138Ø:IF Y>2
   THEN GOSUB 19ØØ
90 IF T(I,1)<5 THEN COLOR 3:DRAW
   "BM"+STR$(T(I,9)*8)+","+STR$
   (T(I,8)*8):UU=VAL(MID$(U$,I,1)):A$=
   UC$(UU):GOSUB 3ØØØ
100 NEXT I
110 FOR E=9 TO 16
120 IF T(E,1)<4 THEN GOSUB 214Ø
130 NEXT E
140 GOSUB 1Ø2Ø
150 GOSUB 229Ø
160 IF VC<>1 AND DE<>1 THEN 6Ø
170 GOSUB 238Ø
180 GOSUB 241Ø:CLS:END
190 REM INITIALIZATION
200 VC=Ø:DE=Ø
2410 REM DELAY
2420 DRAW"BM8Ø,176":A$="PRESS ANY
   KEY":GOSUB 319Ø
2425 G$=INKEY$:IF G$="" THEN 2425
2426 LINE(Ø,176)-(255,183),PRESET,BF
2430 RETURN
```

BBC owners with a disk filing system should type the following before they want to RUN the saved program:

```
*TAPE
FORA%=ØTO&198Ø:?(&EØØ+A%)=
?(PAGE+A%):NEXT
PAGE=&EØØ
OLD
```

First, all except the Commodore and Acorns CLEAR some memory space. The Acorn and Dragon/Tandy set up the graphics mode. Routines to initialize the game and draw the map are then called.

The main loop begins at Line 6Ø and ends at Line 16Ø. It cycles through eight requests for the player to give orders, eight calls to the computer's order selecting routines—one for each unit on the field. It then has 16 calls to the routine that allows the orders to take effect, followed by one call to the test for victory or defeat. Finally, the victory message is printed if the victory test was found true.

At the end of the main loop is a short routine (starting at Line 241Ø) which is used to pause the game at appropriate moments.

PLAYING INSTRUCTIONS

Immediately you RUN the program the computer gets on with drawing the map. The terrain symbols appear first, followed by the opposing units, and the border.

Now's the time to start building your strategy. A series of prompts appear in the text 'window'. Starting from unit one, the unit number and description—for instance, knights—along with the current orders—halt, perhaps. The player is asked Change (Y/N)?

If the answer is Y, a menu of order options is displayed: Fire, Halt or Move. The Fire option is only open to the archers, so any attempt to make another type of unit fire will make the message 'No Bows' appear, and the machine will wait for another choice. If the Move option is chosen, the prompt 'Which way? (N,S,E,W)' appears, ready for the player's choice.

This process is repeated for each unit—the one that's highlighted is the one you have to make the decision on.

THE AFTERMATH

At this stage, Cavendish Field is about as simple as a wargame can be, but still be playable. There are numerous opportunities to tinker with various aspects of the program.

In the final part you will see how the computer can be made into a cleverer opponent, but before you add the new routines, try playing the game as it stands. Beginners, in particular, can get to grips with the strategies involved in wargaming.

PUTTING TOGETHER PASCAL

To many people, the most viable alternative to BASIC as a general purpose language is Pascal, in which powerful structures encourage good programming habits

For those raised on one language, learning another may come easily, or it may require a more dedicated effort. The difference isn't only down to the individual—it also has to do with the relationship between the two languages.

If the structure and syntax of the two languages are similar, it becomes simply a matter of translation—of learning a new vocabulary but then applying the same rules. But there is also the case where the difference between the languages is such that to some extent you have to forget all the rules you already know and apply a new thought process.

The first language covered in this course, LOGO, was designed to be easily accessible for beginners in computing. But if you were brought up on BASIC—almost inevitably the case for home computer owners—it may be much harder to adapt. Sometimes, this can be so much so that it seems difficult to understand the claims that LOGO is easy to learn!

Fundamentally, this is because of the difference in thought processes required for LOGO, since the vocabulary is much simpler than BASIC. LOGO is a much more structured language than most forms of BASIC, and it encourages you to adopt structured thought patterns. (Of course, BBC BASIC has the capability for equally structured forms, but there is no particular need to adopt them at an early stage as there is in LOGO.)

The next language in the course is Pascal, in which structures form an even more important part. In fact, the language cannot exist without them. For the BASIC programmer who is used to going straight to the keyboard and gradually developing a program by adding a little bit here and changing a little bit there, this can seem completely alien—although if you have adopted structured habits in BASIC it will prove far easier to adapt.

So why learn Pascal? Apart from forcing you to adopt good programming habits, it can have certain advantages over BASIC. Its main use is in longer programs, where it can be significantly shorter than BASIC—and much quicker. It is particularly good for number-crunching and business-type applications programs.

THE BACKGROUND TO PASCAL

Pascal was invented in 1970 by Professor Niklaus Wirth of the Federal Institute of Technology in Zurich, and named after Blaise Pascal, one of the great mathematicians and philosophers of the seventeenth century.

Wirth's idea was that Pascal should be used to teach students to program using fundamental concepts of structure. The language itself encourages the programmer to think about the structure of the program before starting to write. And Wirth built into it a range of features that allows the programmed solution to contain the structure of the information that is to be processed.

In other words, writing a program in Pascal means that you plan to solve the problem before you start working on the computer. The planned solution is then refined and refined into smaller and smaller details, until each stage in the process approximates to a procedure which can be programmed. Then, and only then, is the whole program entered.

The concept of Pascal was not totally new, as Wirth took many of the ideas from a language commonly used in universities and government research establishments, known as Algol 60. But whereas Pascal started out in a similar environment, teaching structured programming on large mainframe computers, it has progressed to being widely used on many mini and micro-computers, and is now extensively used for writing commercial packages.

PASCAL AND THE MICRO

Pascal is available in several versions to suit each of the home computers covered here, ready to be loaded from tape or disk. These are available from normal software outlets, although being specialised, may not be held in stock by all dealers—especially those who deal mainly in games. If you don't have the software, don't worry, since you will still be able to understand what is involved even if you cannot run the programs. After all, much of the point of Pascal is that you do the majority of writing away from the machine.

There is an important difference between Pascal and BASIC apart from the way in which it is understood by the computer. It is this which helps to make it faster than BASIC, and also makes it particularly suitable for microcomputers in which the memory is necessarily limited.

One of the disadvantages of BASIC is that it is normally an interpreted language. This means that as each instruction is met, that instruction has to be converted into machine code and then executed. So for a single loop such as:

```
10 FOR N=1 TO 100
```

```
20 PRINT N
30 NEXT N
```

each statement in the loop is converted into machine code one hundred times. And with BASIC, the interpreter must remain resident in memory (usually in ROM) all the time, taking up valuable space.

Pascal, on the other hand, is usually a *compiled* language. When the program has been written, it (the source code) is translated once, and only once, into machine code (the object code). When this has been done, the compiler is no longer needed. This means that it can be swapped in and out of memory onto disk or cassette as required. When the program has been compiled, the memory space previously occupied by it is released as additional working area. (In the case of the BBC and Electron, the compiler relinquishes the high resolution graphics area of memory which it borrows.)

A compiled language is much faster than an interpreted one because it only has to do the translation once, rather than many times. While BASIC compilers do exist, the language is not designed for compilation, and so the machine code that results from compiling a BASIC program is often so long that the finished program may be nearly as slow and large as if it were interpreted. Pascal, on the other hand, is designed for the job. At this stage, Pascal programming is similar to writing a machine code program in assembly language where the completed source code is assembled to give you the machine code. The difference is that designing a Pascal program is much, much simpler than writing in assembly language.

Pascal is thus an effective compromise. But long before you reach the stage of trying to run the program, it has to be designed.

ALGORITHMIC DESIGN

Structured programming in BASIC is covered on pages 173 to 178 and 201 to 207.

But even at its most structured (BBC BASIC) the language is not particularly strong in structures.

The most important lesson to learn in Pascal is to stay away from the machine during the initial design stage. To do this, you need some aids to planning the program. Flowcharting is certainly one way of helping, but flowcharts are themselves unstructured and should only be used sparingly. Another, and better, technique of program design is to use an *algorithm*—a step-by-step method of solving a problem.

A familiar example of an algorithm is a recipe in a cookery book. The printed recipe is like a finished program—it contains all the information necessary for a cook to be able to make the dish. In the same way, a complete program contains all the information the computer needs. But this is many stages down in the development of the algorithm. When you first sit down to design your program, you will have much less definite ideas. So let's look at how you might put together a recipe. Suppose you want to make some mince pies.

Your first attempt to break down the task (the initial algorithm) might be quite simple:

1. Make the pastry
2. Make the filling
3. Shape the pies
4. Cook them

An expert pie-maker might not need all these steps, but the computer is very definitely at the level of an apprentice cook on the first day, so more information is needed.

The next stage is to break each step down further—a process known as stepwise refinement. You will meet this concept again, particularly when you are designing larger programs that cannot be conceived immediately or in one go. In the example above, you could further refine step 1 as:

1a. Weigh flour
1b. Weigh fat
1c. Mix fat with flour
1d. Add water

and each of these steps could be further broken down until you reach a level of simplicity at which the recipe is possible to follow even when the cook is as uniformed as your computer.

The important point in this process is that you have considered the problem first, and not just started on the task randomly. In terms of Pascal programming, the method is very similar. At this stage, you would write each step in ordinary English. Each step is then refined further and further refined until it ends up as a Pascal statement. Each step in the initial algorithm may well represent a different procedure or function in the finished program—but in a complex program the process of refinement may mean that each step actually embraces a whole set of procedures by the time the program is completed.

So far, so good. But how can you end up designing a Pascal program unless you know what Pascal looks like, and what statements are accepted. The answer is to look at some examples.

A PASCAL PROGRAM

One of the characteristics of Pascal programs is that they often tend to look unnecessarily complicated when dealing with a simple task. For example, look at the following program to add two numbers and print the result. It is much longer than the BASIC equivalent, although it will still run faster. Don't be put off by appearances, however. Once you start dealing with more complicated tasks, Pascal becomes more and more economical compared to the equivalent BASIC:

```
program example1(input,output);
var no1,no2,result:integer;
begin
  read(no1,no2);
  result: = no1 + no2;
  writeln(result)
end.
```

This program could result from an initial algorithm something like:

1. Set up initial conditions
2. Input numbers
3. Add numbers
4. Print result
5. Stop

As you can see, the final program is virtually the same as this algorithm. One of the reasons why short Pascal programs are longer than the equivalent BASIC is that you must give steps 1. and 5., neither of which is needed in BASIC. You have to tell the computer what is involved in the program, tell it to start and tell it to finish.

Let's take the Pascal line by line:

```
program example1(input,output);
```

means you are going to read and write within a program called example1. If the program did not require input data you could just have

```
Program example1(output);
```

The semi-colon in both these lines indicates the end of a statement.

All variables in Pascal must be declared as a particular type. The next line says that the variables no1, no2 and result represent whole numbers; that is of type integer. Again the semi-colon tells the computer that this is the end of that statement.

```
var no1,no2,result:integer;
```

Now you come to the main block of the program. The statements begin (note there is no semi-colon as this is the start of *multiple*, or compound, *statement*) and end. will occur at the beginning and end of every Pascal program. The full stop after end indicates that it is the very last statement of this particular program and signals a halt.

BASIC is made easier to follow by using comments in the form of statements. This is true of any programming language irrespective as to whether it is high or low level. Pascal is no exception.

In Pascal any line can be commented on by using curly brackets { and } or if your machine does not have these (* and *). The BBC/ Electron S-Pascal compiler will accept only { and } which in fact are displayed on the screen as $\frac{1}{4}$ and $\frac{3}{4}$.

read (no1,no2); is very similar to the BASIC statement INPUT NO1,NO2. You are allowed to use no1 and no2 as variable names because they have already been declared in the var statement.

```
result : = no1 + no2;
```

is read as result becomes no1 + no 2 in much the same way as in BASIC. Strictly speaking this could read let result = no1 + no2 to show that we are summing the values of the variables no1 and no2 and assigning this value to a variable called result.

Another bit of programming:

```
writeln(result)
```

has a similar effect to the BASIC statement PRINT RESULT, that is, it will output the value of the variable result. Notice how there is no ; after this statement. This is because a semi-colon before an end is unnecessary as the end itself tells us the end of a statement has been reached. If a semi-colon is put in, most compilers will allow it by inserting a blank line before the end.

PROGRAMS AVAILABLE

There are Pascal programs available for all the machines covered by *INPUT*, though some are less comprehensive than others—particularly the version for the Dragon 32.

One of the by-products of the fact that Pascal is a compiled language is that this may affect your ability to edit it. In some versions, it is important to get the code right first time, since once compiled it is inaccessible for editing. In other versions, the program may be compiled each time it is run, in which case the source code is retained in memory and can be corrected with the usual BASIC editor. In the latter case, a small area of memory is reserved for the program.

Pascal programs can normally be written in either upper or lower case, but lower case is used throughout this series of articles to enable the programs to run the Pascal package for Acorn machines.

The Pascal program available for the Spectrum has a line editor built in, a random number function and supports graphics.

Commodore Pascal makes use of many of the 64's standard features, though not everything is covered, there's no sprite handling for example. Part of the Commodore package is an editor which behaves very much like the BASIC editor but is enhanced with auto line numbering and renumbering, FIND lists all occurrences of a specified string and REPLACE substitutes one string for another.

The S-Pascal program available for Acorn machines is very adaptable and the facilities offered by the editor are extremely sophisticated. As well as full screen editing, it also allows block copy, moves and deletions and various search and/or replace operations.

In the next part of *INPUT* we will continue with our look at Pascal and examine some of the procedures available in the language and how they compare to the procedures found in BASIC.

We will also take a look at how Pascal programs are compiled and show examples of how when you come to write a program in this particular language, you must work within a precisely defined form for each statement.

MATCH THAT! A COMPUTER PUZZLE SETTER

Here's a quick recap of the rules in case you've forgotten them or in case you haven't played the game before. The object of the game is to find out the colours of the four counters chosen by the computer, in the least number of guesses. The colours have to be in the right order too.

Input your guess by typing in the initial letters of the colours, for example, R B B Y for red, blue, blue and yellow or, on the Commodore, a number from 1 to 6 to match the colours on those keys. The computer then responds by giving you a few cryptic clues telling you how many colours are correct and how many are in the right position—although it doesn't tell you which colours are the right ones! The code it uses is a white dot for a correct colour in the wrong place a black dot for a correct colour that's also in the right position. Use these clues to decide on your next guess. You have 12 goes to find the correct code.

The colours used by each of the programs are Spectrum and Acorn: yellow, white, blue, red, cyan and magenta, Commodore and Vic; the colour keys from 1 to 6, Dragon and Tandy: yellow, blue, red, cyan, mauve and orange.

```
10 BORDER Ø: INK Ø: PAPER 4: CLS : LET
   N$="671254": DIM C(4): DIM G(4): DIM
   F(4,2): LET C$="YBRCM"
14 PRINT AT 16,Ø;"COLOURS: — "''"Y=
   YELLOW B=BLUE□C=LIGHT BLUE□
   □□R=RED□□□□M=MAUVE□W
   =WHITE."
15 FOR N=USR "A" TO USR "A"+7: READ
   A: POKE N,A: NEXT N
17 DATA Ø,24,6Ø,126,126,6Ø,24,Ø
2Ø FOR K=1 TO 4: LET C(K)=VAL N$(INT
   (RND*5)+1): NEXT K: LET G=1
3Ø INPUT "WHAT'S YOUR GUESS ?□";B$
35 IF LEN B$<>4 THEN GOTO 3Ø
9Ø PRINT AT G,Ø;"GUESS□No ";G;AT G,14;:
   FOR K=1 TO 4: LET
   G(K)=(7*(B$(K)="W"))+(6*(B$(K)=
   "Y"))+(B$(K)="B")+(2*(B$(K)=
   "R"))+(5*(B$(K)="C"))+(3*(B$(K)
   ="M"))
92 IF G(K)=Ø THEN LET K=4: NEXT K:
   GOTO 3Ø
95 PRINT INK G(K); BRIGHT 1;CHR$ 144;: IF
   K<>4 THEN PRINT BRIGHT 1;"□";
97 NEXT K
1ØØ PRINT AT G,24;
11Ø LET N=Ø: LET R$="□": FOR K=1
   TO 4: LET F(K,1)=Ø: LET F(K,2)=Ø: IF
   G(K)=C(K) THEN LET
   R$=R$+"□"+CHR$ 16+CHR$
   Ø+CHR$ 144: LET F(K,1)=1: LET
   F(K,2)=1: LET N=N+1
12Ø NEXT K
13Ø FOR K=1 TO 4: IF F(K,1)=1 THEN
   GOTO 17Ø
14Ø FOR J=1 TO 4: IF F(J,2)=1 THEN
   GOTO 16Ø
15Ø IF C(J)=G(K) THEN LET
   R$=R$+"□"+CHR$ 16+CHR$
   7+CHR$ 144: LET F(J,2)=1: LET J=4
16Ø NEXT J
17Ø NEXT K
18Ø PRINT AT G,23;R$: INK Ø: IF N=4 THEN
   PRINT AT 21,Ø;"YOU GOT THE CORRECT
   CODE IN□";G'"GUESSES.": GOTO 23Ø
19Ø LET G=G+1: IF G<13 THEN GOTO 3Ø
2ØØ PRINT "THE CORRECT CODE WAS□";:
   FOR K=1 TO 4: PRINT INK C(K);CHR$
   144;"□";: NEXT K
22Ø PRINT
23Ø PRINT "LIKE ANOTHER GAME ?"
24Ø LET A$=INKEY$: IF A$="" THEN
   GOTO 24Ø
25Ø IF A$="Y" THEN RUN : STOP
```

For the Vic change Line 24 to POKE 36879, 11Ø.

```
1Ø C$="123456":D$="▣▣▣▣▣▣
   ▣▣▣▣▣▣▣▣":K$="◘◘
   ◘◘"
2Ø FOR K=1 TO 4:C(K)=INT(RND(1)*6)
   +1:NEXT K
24 POKE 5328Ø,6
25 G=1:PRINT "♥π▟▟GUESS THE
   CODE"
26 PRINT "†<ENTER FOUR
   NUMBERS":PRINT"▟▟BETWEEN 1 & 6)"
3Ø PRINT "▤▒▣▣"LEFT$(D$,G):PRINT
   G;:INPUT B$
4Ø PRINT"○"SPC(1Ø)"▒◫";:N=Ø:
   BB$="":N1=Ø:N2=Ø
5Ø FOR K=1 TO 4:CC(K)=C(K):NEXT K
6Ø FOR K=1 TO 4:IF
   VAL(MID$(B$,K,1))=C(K) THEN
   CC(K)=255
7Ø NEXT K
1ØØ FOR K=1 TO 4:XX= VAL(MID$(B$,
   K,1))
11Ø  IF  XX=C(K)  THEN  N1=N1+1:
   N=N+1: GOTO  14Ø
12Ø  FOR  KK=1  TO  4:IF  XX=CC (KK)
   THEN  N2=N2+1: CC(KK)= 255: GOTO
   14Ø
125  NEXT  KK
14Ø NEXT K:PRINT "■" LEFT$(K$,N1)
   "▬" LEFT$(K$, N2) TAB(15)
   "▒▬◫";
15Ø FOR K=1 TO 4: POKE 646, ABS(VAL
   (MID$(B$, K, 1))—1): PRINT "●";:
   NEXT  K: PRINT  "▒◫"
18Ø IF N=4 THEN 2ØØ
19Ø G=G+1:IF G=13 THEN 21Ø
195 GOTO 3Ø
2ØØ PRINT "▣◣YOU GOT THE
   CORRECT":PRINT"CODE
   IN"G"GUESSES":GOTO 23Ø
21Ø PRINT "▣◣I WIN, THE
   CORRECT":PRINT"CODE WAS ▟■";
22Ø FOR K=1 TO 4:PRINT CHR$(48+C(K))
   ;:NEXT K:PRINT
23Ø PRINT "†ANOTHER GAME ?(Y/N)"
24Ø GET A$:IF A$<>"Y" AND
   A$<>"N" THEN 24Ø
25Ø IF A$="N" THEN PRINT "♥":END
255 GOTO 2Ø
```

```
1Ø MODE2:COLOUR13Ø:CLS:C$="R*YB
   MCW":DIMC(4) ,G(4) ,F(4,2)
15 VDU23,224,Ø,24,6Ø,126,126,6Ø,24,Ø
2Ø FORK=1TO4
22 C(K)=RND(7):IF C(K)=2 THEN 22
25 NEXT:G=6
3Ø COLOUR7:INPUTTAB(Ø,29)"ENTER
   GUESS□",B$:PRINTTAB(Ø,29)SPC
   (39);
35 IF LENB$<>4THEN3Ø
9Ø E$="":FORK=1TO4:D$=MID$
   (B$,K,1):G(K)=INSTR(C$,D$)
92 IFG(K)=Ø THEN K=4:NEXT:GOTO3Ø
95 E$=E$+CHR$17+CHR$(G(K))+
   CHR$224
```

Let your computer challenge you to this classic game of logical thinking. See if you can crack the computer's cryptic colour code and select the correct sequence

- THE RULES OF THE GAME
- CHOOSING THE COLOURS
- THE COMPUTER'S REPLY
- WORKING OUT THE RIGHT STRATEGY

```
97 NEXT:PRINTTAB(Ø,G);G – 5;TAB(5,G)E$
100 PRINTTAB(5,G);
110 N = Ø:R$ = "□":FOR K = 1TO4:
   F(K,1) = Ø:F(K,2) = Ø:IF G(K) = C(K)
   THEN R$ = R$ + CHR$17 + CHR$Ø +
   CHR$224:F(K,1) = 1:F(K,2) = 1:N = N + 1
120 NEXT
130 FORK = 1TO4:IF F(K,1) = 1THEN 170
140 FORJ = 1TO4:IF F(J,2) = 1THEN 160
150 IFC(J) = G(K) THEN R$ = R$ +
   CHR$17 + CHR$7 + CHR$224:F(J,2) = 1:
   J = 4
160 NEXT
170 NEXT
180 PRINTTAB(14,G)R$:IF N = 4THEN PRINT
   "YOU GOT IT IN□";G – 5:GOTO 220
190 G = G + 1:IF G < 17  THEN  30
200 PRINT"I  WON"
210 PRINT"THE CODE WAS□";:FOR K = 1
   TO4:COLOURC(K):VDU224,32:NEXT
220 PRINT'"LIKE ANOTHER GAME ?"
230 A$ = GET$
240 IF A$ = "Y" THEN RUN
```

```
10 DIMC(3),G(3),F(3,1):C$ =
   "YBRCMO"
20 CLS:FORK = ØTO3:C(K) = RND(6):
   NEXT:G = 1:PRINT@9,"guess the code"
30 PRINT@416,"INPUT GUESS NO";G;"?□
   □(EG RBYC)":PRINT@448:B$ = ""
40 A$ = INKEY$:IF A$ = "" THEN 40
50 IF A$ = CHR$(13) AND LEN(B$) = 4 THEN
   90
60 IF A$ = CHR$(8) AND LEN(B$) > Ø THEN
   B$ = LEFT$(B$,LEN(B$) – 1)
70 IF LEN(B$) < 4 AND INSTR(C$,A$) < >
   Ø THEN B$ = B$ + A$
80 PRINT@448,B$:GOTO40
90 PRINT@32*G,"GUESS NO";G;:FORK = Ø
   TO3:G(K) = INSTR(C$,MID$(B$,K + 1,1)):
   C = G(K) – (G(K) > 3)
100 PRINT@32*G + 11 + K*2,CHR$(143 +
   C*16);:NEXT
110 N = Ø:R$ = "□ □":FORK = ØTO3:
   F(K,Ø) = Ø:F(K,1) = Ø:IFG(K) = C(K)THEN
   R$ = R$ + "□" + CHR$(128):F(K,Ø) = 1:
   F(K,1) = 1:N = N + 1
120 NEXT
130 FORK = ØTO3:IF F(K,Ø) = 1 THEN 170
140 FORJ = ØTO3:IF F(J,1) = 1 THEN 160
150 IF C(J) = G(K)THENR$ = R$ +
   "□" + CHR$(207):F(J,1) = 1:J = 3
160 NEXT
170 NEXT
180 PRINTR$:IF N = 4 THEN 200
190 G = G + 1:IF G = 13 THEN 210
   ELSE 30
200 PRINT@416, "□YOU GOT THE
   CORRECT CODE IN";G;"□GUESSES":
   GOTO 230
210 PRINT@416,"□I WIN, THE CORRECT
   CODE WAS"
220 FORK = ØTO3: PRINT@448 + K*2,"□";
   CHR$(143 + C(K)*16 – 16*(C(K) >
   3));:NEXT
230 PRINT:PRINT" DO YOU WANT ANOTHER
   GAME ?(Y/N)"
240 A$ = INKEY$:IF A$ < > "Y" AND
   A$ < > "N" THEN 240
250 IF A$ = "N" THEN CLS:END ELSE20
```

FILE IT DON'T FORGET IT

Open up a channel to your tape recorder or disk drive and direct your valuable data down the lines. Knowing how to handle files is the key to many a program

Any program that handles a lot of data must have some convenient way of storing it. In an earlier article, on page 105, there is a short program showing how a simple telephone directory can be stored in DATA statements. The program showed how an item of data can be searched for and printed out, but there was no way of updating the list without BREAKing into the program and altering the program lines themselves. Also, it is only possible to save the information with the program itself. The program is thus tied to whatever information it contains, which is not very satisfactory.

It is obviously much better to store data separately on tape or disk rather than in the program. The information can be read into the computer as easily as reading in the lines of a program, but items can be deleted or added much more easily—and the person using the program doesn't have to know anything about programming to do so.

The data is stored on tape or disk as a *file* and many of the Applications programs in *INPUT* store the data in this way. The Hobbies file, Home Finance, Calendar and Spreadsheet programs are just a few that rely on being able to handle large amounts of information. If you look closely at these programs you should be able to trace the part that saves and loads the data. It is a small part of the program in each case, and a great deal more goes into things like memory allocation, prompts, displays and such like.

By storing the information separately from the 'core' program, you have what amounts to a virtual memory: the basic program can be used for any number of different subject files, each with a different number of records or fields, tailored to the purpose of each file.

Sequential data storage like this is a favourite method. The programming is straightforward and the principles are well understood. Also, the data is convertible to a form that can actually be passed from one program to another or, indeed, from one computer to another.

Where a sequential file varies from others is that all data is stored *serially*, with each item being separated only by a single byte. All the information is collected in memory before being written out to a file for storage.

The first step in saving a sequential file is to write information to your storage device. This is done using your computer's form of the OPEN command. All of the most commonly used commands were described in the introductory article on pages 622 to 627.

This article explains in detail how to set up your own files on tape and disk, so you can incorporate them into your own programs.

CREATING A FILE

The program shows how to set up a more sophisticated version of the telephone directory file mentioned earlier. But you could easily change the prompts and arrays to accept other types of information.

Entries are made under first name, second name and number. Because telephone numbers will contain spaces and possibly dashes as well, they are best treated as a string. With other types of data you might want to have some string arrays and some numerical arrays. It doesn't matter what type of data is saved, or in which order, as long as it is read back in the same order and into the identical sort of variable.

It is possible to send or write information straight into a file but it is more usual, and more convenient, to store it in an array first and then save the whole array.

The first part of the program sets up a simple loop so you can enter the data. Enter as many names and numbers you like—up to the number dimensioned in Line 1Ø. Change the dimension if you want more than 50. When you've finished, press ENTER or RETURN instead of the next name and number and the INPUT routine will stop.

The second half, from Line 1ØØ onwards, writes the data to a file. This part is explained after each program.

```
1Ø DIM A$(5Ø,15): DIM B$(5Ø,15): DIM
   T$(5Ø,12): DIM N(1)
2Ø LET N=Ø
3Ø LET N=N+1
4Ø INPUT "FIRST NAME ";A$(N)
5Ø INPUT "SECOND NAME ";B$(N)
6Ø INPUT "TELEPHONE NUMBER ";T$(N)
7Ø IF A$(N)< >"□□□□□□□□□
   □□□□□□"AND N<5Ø THEN
   GOTO 3Ø
8Ø CLS: PRINT "SAVING DATA NOW"
```

Use this section if you are saving the data to tape:

```
1ØØ SAVE "COUNT" DATA N()
11Ø SAVE "F.NAMES" DATA A$()
12Ø SAVE "S.NAMES" DATA B$()
13Ø SAVE "T.NUMBS" DATA T$()
14Ø PRINT "DATA SAVED"
15Ø STOP
```

Use the following section if you have a Microdrive:

```
9Ø INPUT "ENTER DRIVE NUMBER ";DRV
1ØØ SAVE *"M";DRV;"COUNT" DATA N()
11Ø SAVE *"M";DRV;"F.NAMES" DATA A$()
12Ø SAVE *"M";DRV;"S.NAMES" DATA B$()
13Ø SAVE *"M";DRV;"T.NUMBS" DATA T$()
14Ø PRINT "DATA SAVED"
15Ø STOP
```

On the Spectrum the first section of program from Lines 1Ø to 8Ø is common to both tape and Microdrive, however, the second part that saves the data is different. Both versions save the data as an array, but with a Microdrive you have to find out which drive is being used (there may be more than one) and add *"M";DRV; after each SAVE command. Note that each array must be given a file name before it can be saved, and the instruction DATA must appear before each array.

Type in the appropriate version of the program and then try entering some data and saving it ready for the next program.

There is an alternative way of saving the data when using a Microdrive. The problem with the last method is that the whole array must be saved even if only a part of it is filled up. This wastes precious space on your Microdrive tape. The alternative version, below, saves each item of the array separately and stops when the data runs out.

```
1ØØ INPUT "ENTER DRIVE NUMBER ";DRV
11Ø OPEN #4;"M";DRV;"FILE"
12Ø PRINT #4;N
125 FOR L=1 TO N
```

- ADVANTAGES OF STORING DATA IN A FILE
- TAPE, DISK AND MICRODRIVE FILES
- OPENING AND CLOSING A FILE
- READING AND WRITING
- DETECTING AN END-OF-FILE MARKER
- DUMMY TERMINATORS
- USING THE DATA

```
130 PRINT #4;A$(L)'B$(L)'T$(L)
140 NEXT L
150 CLOSE #4
160 PRINT "DATA SAVED"
170 STOP
```

As before, you first have to type in the drive number. Line 11Ø then opens a channel to the Microdrive. Any channel from 4 to 15 is available (channels Ø to 3 are reserved for the screen and the ZX printer) and channel 4 is the one most commonly used. Line 12Ø then writes the value of N—the number of data items—to the file through this channel, and Line 13Ø writes each item of data in the arrays, one at a time, until the maximum number N is reached. CLOSE #4 then closes the channel.

It is very important to type in these lines exactly as shown. Semi-colons must be used to separate the items in Lines 11Ø and 12Ø, and in Line 13Ø there must be a semi-colon after PRINT #4 and then the other items must be separated by a single apostrophe.

If you want to try out this program, SAVE the other version first, type in the new lines and then try saving the data again, preferably on a separate tape so you don't get mixed up. The data should still be stored in the arrays, so take care not to type NEW!

```
10 DIM A$(5Ø),B$(5Ø),T$(5Ø)
3Ø N=N+1
4Ø INPUT "■■FIRST NAME□■";A$(N)
5Ø INPUT "■SECOND NAME■";
   B$(N)
6Ø INPUT "■TELE NUMBER■"; T$(N)
7Ø IF A$(N)< >"" AND N<5Ø THEN 3Ø
8Ø PRINT "♥SAVING DATA NOW"
1ØØ OPEN 1,1,1,"FILE"
11Ø PRINT#1,N-1:Q$=CHR$(34):
   C$=","
12Ø FOR Z=1 TO N-1
13Ø PRINT#1,Q$A$(Z)Q$C$Q$B$(Z)Q$C$
   Q$T$(Z)Q$
14Ø NEXT Z
15Ø CLOSE 1
16Ø PRINT "DATA SAVED"
17Ø END
```

For disk files change Line 1ØØ to:

```
1ØØ OPEN 1,8,1, "FILE"
```

The file is opened by specifying the file number, device number and address, followed by the file name in quotes. (See the box below for more information on the exact form of the OPEN command.) Only one file is used by this program so the file number is 1. The device number is 1 for tape and 8 for disk drive. The address is 1 which specifies that the program is *writing* to a file.

PRINT#1 means 'write the next item of information to file 1' so the first part of Line 11Ø saves the value of N-1 which will be used later when reading back the data.

The remainder of Line 11Ø sets Q$ equal to a double quote mark (character 34) and C$ equal to a comma. These are then used in Line 13Ø which writes the names and numbers to the file. The commas are used to separate each item of data—other separators you could use are the colon, semi-colon or a RETURN code (CHR$(13)). The quotes are used to surround each data string. The quotes are a safety measure in case the original data contained any punctuation which the program would take as a separator.

The CLOSE 1 instruction in Line 15Ø closes the file.

```
1Ø DIM A$(5Ø),B$(5Ø),T$(5Ø)
2Ø N=Ø:REPEAT
3Ø N=N+1
4Ø INPUT"FIRST NAME ",A$(N)
5Ø INPUT"SECOND NAME ",
   B$(N)
6Ø INPUT"TELEPHONE NUMBER", T$(N)
```

COMMODORE FILE COMMANDS

The Commodore 64's range of file handling activities is much more complex than the other computers' and is dictated by the exact form of the OPEN command employed. It's worth having an overall look at this before looking at the programs in this article. It takes the general syntax:

```
OPEN lfn, dn, sa,"filename,type,
mode"
```

Here, lfn stands for *logical file number*, dn stands for *device number*, sa for *secondary address*. The file name is followed by the file type, and the mode of using the channel that's been opened.

To OPEN a sequential file, first decide on the lfn. For 'solo' program applications, normally only one channel need be open at once and more or less any arbitrary lfn may be selected. It can range from 1 to 255 and you can select any number within that range although it's good practice to stick to below 128 because those higher are really designed for use in other ways.

The lfn is accessed by both read and write operations.

The devices which can be specified are:

Ø—Keyboard
1—C2N tape unit
2—RS232 user port (modem, printer etc)
3—Screen (monitor or TV)
4/5—Printer (serial IEEE)
6—Plotter
8/9—Disk units (1541)
1Ø—255 User specified

For file handling, device numbers 1, 8 or 9 are normally all that are ever used. One of the latter two must be used in the command structure if a disk unit is used.

The secondary address, sa, can take a value of between 2 and 14 (15 is reserved for specific disk commands and errors) and indicates the channel over which input or output is to take place.

A maximum of three sequential channels may be OPEN at once if you're using a disk based file handling program. But note that different sa's must be used otherwise the first file will be automatically CLOSEd when another file with the same sa is OPENed.

The sa values normally used are:

Ø—read a file
1—write to file
2—write to file with end-of-file marker

If sa is unspecified default is Ø.

Then, in quotes, follow in turn the filename (which can be up to sixteen characters in length), the file type and the mode.

It's a good idea always to use a file name even though this is not a strict requirement of tape data saves (it's necessary for disk saves, however). There are five different file types to choose from. The program file is the most common and is the default value if you don't bother to specify which type:

S—sequential file
U—user file *
P—program file
R—relative file *
D—delete *

(* indicates disk use only)

```
7Ø UNTIL A$(N)="" OR N=5Ø
8Ø CLS: PRINT "SAVING DATA NOW"
1ØØ C=OPENOUT"FILE"
11Ø PRINT#C,N
12Ø FOR L=1 TO N
13Ø PRINT#C,A$(N),B$(N),T$(N)
14Ø NEXT
15Ø CLOSE#C
16Ø PRINT "DATA SAVED"
17Ø END
```

In this example, the variable C takes the number of the channel created by the OPENOUT statement. There is no need to know which channel has been opened for you as the variable C is used throughout the program. The statement also prepares the tape or disk for the file by recording a header, which includes the file name.

If a disk is used, 64 sectors are immediately reserved when a new file is opened. One possible problem to be aware of when using disks is that the program will overwrite any existing file with the same name unless the old one is locked.

PRINT#C means 'write the next item to the file', so Line 11Ø saves the value of N which will be used when reading back the information. Line 13Ø writes each of the items of data into the file.

The last command parameter, *mode* simply instructs the computer on how to use the channel that's been OPENed using one of the following four key letters:

W—write a file
R—read a file
A—add to a sequential file
M—read unclosed file

All except the first of the command parameters can be omitted, when the OPEN channel defaults to the cassette unit—device 1.

So you can end up with OPEN instructions of many forms. Here are some simple examples:

OPEN 1,1,1,"FILE"—write to tape
OPEN 1,1,Ø,"FILE"—read from tape
OPEN 1,8,1,"FILE"—write to disk
OPEN 1,8,Ø,"FILE"—read from disk
OPEN 1,1,1,"FILE,S,W"—write sequential tape file
OPEN 1,1,1,"FILE,S,R"—read sequential tape file
OPEN 1,8,1,"FILE,S,W"—write sequential disk file

The CLOSE#C instruction closes the channel. If this is not done there may be problems, not least of which is that data may go missing. Always CLOSE a file at the end of a writing routine.

Enter some names and numbers and save the information ready for the next section of program.

```
1Ø DIMA$(5Ø),B$(5Ø),T$(5Ø)
3Ø N=N+1
4Ø INPUT"FIRST NAME□";A$(N)
5Ø INPUT"SECOND NAME□";
  B$(N)
6Ø INPUT"TELEPHONE NUMBER□";
  T$(N)
7Ø IF A$(N)<>"" AND N<5Ø THEN 3Ø
8Ø CLS: PRINT "SAVING DATA NOW"
```

Use this section if you are saving the data on tape:

```
1ØØ OPEN"O",#-1,"FILE"
11Ø PRINT#-1,N
12Ø FORL=1TON
13Ø PRINT#-1,A$(L),B$(L),T$(L)
14Ø NEXT
15Ø CLOSE#-1
16Ø PRINT "DATA SAVED"
17Ø END
```

Use this section if you have a disk drive:

```
1ØØ CREATE"FILE"
11Ø FWRITE"FILE";N
12Ø FORL=1TON
13Ø FWRITE"FILE";A$(L);",";B$(L);",";
  T$(L)
14Ø NEXT
15Ø CLOSE
16Ø PRINT "DATA SAVED"
17Ø END
```

The first section of this program is common, but the second part, that writes data to the file, is different for tape and disk.

To take the tape section first, a file is created using the OPEN"O" command which means open for output. This is followed by #-1 which tells the computer to send the data to a tape recorder, and then the name of the file in quotes.

PRINT#-1 means 'write the next item to tape file', so Line 11Ø saves the value of N—the number of data items—ready to be used when reading back the information. Line 13Ø then writes each item of data to the file—note the commas separating each item. Finally, the file must be closed at the end of the writing routine with the command CLOSE#-1 as shown in Line 15Ø.

Creating a file with a Dragon Data disk drive is broadly similar except that different commands must be used. A file is opened with CREATE followed by the name of the file, in quotes, up to 8 characters. Information is written to the file using FWRITE. Note that the file name must be specified in each writing operation and that a semi-colon is used to separate the items rather than a comma as before. As a further complication, commas have to be specially sent to the file to separate items when there are several on the same line, as shown in Line 13Ø. The simple instruction CLOSE is all that's needed at the end of this program without having to specify a file name.

TROUBLESHOOTER

It is very easy to confuse the OPEN commands on the BBC computer especially as they are different in Basic I and Basic II.
With Basic I:
- OPENOUT opens a new file for output from the computer; it is used to create a new file
- OPENIN opens an existing file for input or output

With Basic II
- OPENOUT opens a new file for output to the computer; this is the same as Basic I
- OPENIN opens an existing file for input to the computer *only*
- OPENUP opens an existing file for input or output and is the same as OPENIN in Basic I

Any program written on a Basic I computer will run in Basic II. But programs written in Basic II must use OPENUP instead of OPENIN if they are also to run in Basic I.

READING THE FILE

The program to read the file back into memory is just a reverse of the save program. There's also a line to display the information on the screen so you can prove to yourself the data really has been read in.

For tape use this section:

```
2ØØ CLEAR
21Ø DIM N(1): LOAD "COUNT" DATA N()
215 DIM A$(N(1),15): DIM B$(N(1),15): DIM
  T$(N(1),12)
22Ø LOAD "F.NAMES" DATA A$()
23Ø LOAD "S.NAMES" DATA B$()
```

```
240 LOAD "T.NUMBS" DATA T$()
250 FOR L=1 TO N(1)
260 PRINT A$(L), B$(L), T$(L)
270 NEXT L
```

For Microdrive use this:

```
200 CLEAR
205 INPUT "ENTER DRIVE NUMBER ";DRV
210 DIM N(1): LOAD *"M";DRV;"COUNT"
   DATA N()
215 DIM A$(N(1),15): DIM B$(N(1),15): DIM
   T$(N(1),12)
220 LOAD *"M";DRV;"F.NAMES" DATA A$()
230 LOAD *"M";DRV;"S.NAMES" DATA B$()
240 LOAD *"M";DRV;"T.NUMBS" DATA T$()
250 FOR L=1 TO N(2)
260 PRINT A$(L),B$(L),T$(L)
270 NEXT L
```

To use this program rewind the tape to the start of the file then type RUN 2ØØ to read in the data, don't RUN the whole program again.

Line 2ØØ is just there to clear the original names and numbers from the computer's memory so the program only prints out those read from the file. Line 21Ø dimensions a new array N() with one element. This is used to store the value of N saved last time. The other arrays are dimensioned in the next line using this value. Since the maximum number of elements are now known it saves reading in empty elements of the array.

The rest of the program is very straightforward, and can be written simply by substituting the command LOAD instead of SAVE. There is no need to close a file opened for reading, only for writing.

Again, there is an alternative version for the Microdrive (try not to get all these versions mixed up). Once more, it is similar to the writing procedure except that PRINT 4# is charged to INPUT#4 and in Line 23Ø the items to be input are separated by semi-colons:

```
200 CLEAR
205 INPUT "ENTER DRIVE NUMBER "DRV:
   OPEN #4;"M";DRV;"FILE"
210 INPUT #4;N: DIM A$(N,15): DIM
   B$(N,15): DIM T$(N,12)
220 FOR L=1 TO N
230 INPUT #4;A$(L);B$(L);T$(L)
235 PRINT A$(L);B$(L);T$(L)
240 NEXT L
250 CLOSE #4
```

```
200 CLR: OPEN 1,1,0,"FILE"
210 INPUT#1,N: DIM A$(N), B$(N), T$(N)
220 FOR Z=1 TO N
230 INPUT#1,A$(Z),B$(Z),T$(Z)
235 PRINT A$(Z),B$(Z),T$(Z)
240 NEXT Z
250 CLOSE 1
```

Again, for disk files change Line 2ØØ to:

```
200 OPEN 1,8,0, "FILE"
```

To use this program rewind the tape to the start of the file then type RUN 2ØØ to read in the data, don't RUN the whole program again.

The CLR instruction in Line 2ØØ simply wipes out all the original names and numbers so you can be sure that the information printed out really has been read from the file. The rest of the line opens the file for reading—note that the third number after OPEN has changed from a 1 to a Ø.

Line 21Ø inputs the first item of data. This is the number of items in the array and is used to set the limit of the loop that reads in the data. Once N is known, the arrays are dimensioned to the correct size. Line 23Ø then reads in each string from the file and the next line prints them out on the screen. Again, the file must be closed with CLOSE 1.

```
200 CLEAR:C=OPENIN"FILE"
210 INPUT #C,N:DIM A$(N),B$(N),T$(N)
220 FOR L=1 TO N
230 INPUT#C,A$(L),B$(L),T$(L)
235 PRINTA$(L),B$(L),T$(L)
240 NEXT
250 CLOSE#C
```

To use this program rewind the tape to the start of the file and then type GOTO 2ØØ to read in the data, don't RUN the whole program again.

The CLEAR instruction in Line 2ØØ is there simply to wipe out all the original names and numbers in the computer's memory so you can see that the information printed out by Line 235 really has been read from the file. The rest of Line 2ØØ opens the file for input to the computer. It is very easy to get these commands mixed up, especially if you have Basic II, so have a look at the Troubleshooter on page 1361 if you are confused.

Line 21Ø inputs the first item of data—N—which is the number of items in the arrays. You can now see why the variable N was saved in the first procedure, it is used here to control the loop in the reading procedure. Once N is known, the arrays are dimensioned to the correct size. Line 23Ø then reads in each piece of information and the next line prints them out on the screen. Again, the file must be closed using CLOSE#C.

For tape use this section:

```
200 CLEAR 2000:OPEN"I",# -1,"FILE"
210 INPUT# -1,N: DIM A$(N), B$(N),
   T$(N)
220 FOR L=1 TO N
230 INPUT# -1,A$(L),B$(L),T$(L)
235 PRINTA$(L);B$(L);T$(L)
240 NEXT L
250 CLOSE# -1
```

For disk use this:

```
200 CLEAR 2000
210 FREAD"FILE",FROM0;N: DIM A$(N),
   B$(N), T$(N)
220 FOR L=1 TO N
230 FREAD"FILE";A$(L),B$(L),T$(L)
235 PRINTA$(L);B$(L);T$(L)
240 NEXTL
250 CLOSE
```

To use this part of the program first rewind the tape to the start of the file and then type RUN 2ØØ, don't RUN the whole program again.

Line 2ØØ simply clears the original variables so you can be sure that the names and numbers printed out here actually come from the file and not from memory.

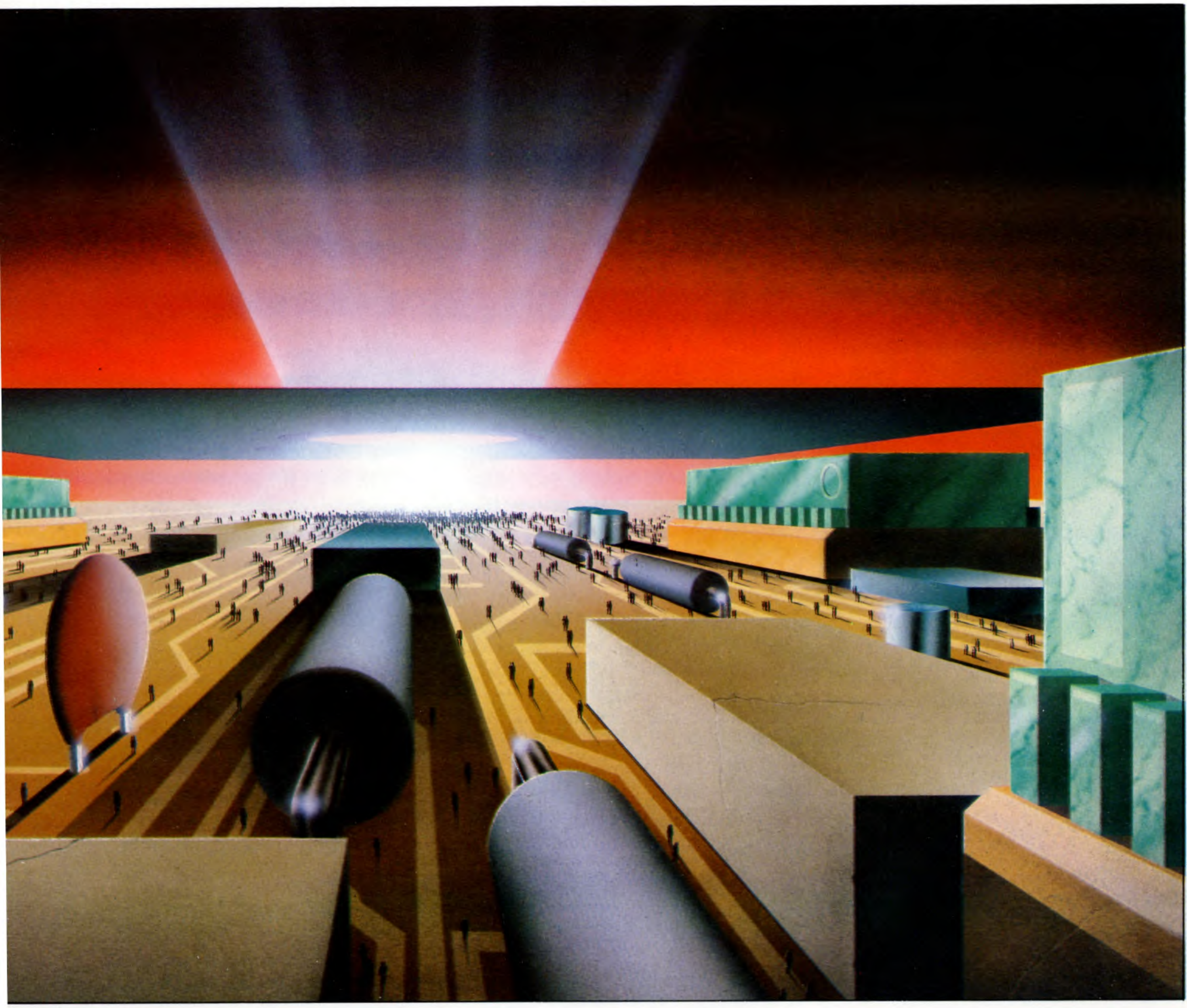
Again, the two versions are different in detail. The tape version opens the file for input using OPEN"I" followed, as usual, by # −1 and the name of the file. Line 21Ø inputs the first item of data—N, the number of records—and this number is used to dimension the arrays to the correct size. This variable also controls the loop which reads in the rest of the data. Note that the array variables are in the same order in which they were saved and that they are separated by commas. As usual, the file has to be closed when you've finished reading.

The disk version is similar in structure except there is no need to use a special command to open the file. Data is read using FREAD followed by the file's name. Line 21Ø shows the correct way to read in the first item of data. The FROMØ instructs the computer to start at the beginning of the file—from byte Ø. You can specify any position although this is only useful when using random access file. Serial files, like the one you have just set up, always start from Ø. The rest of the data is read by Line 23Ø. Have a close look at the way this line is made up—a semi-colon must precede the first item, but the other items are separated by commas.

THE END OF THE FILE

The last program used the loop variable N to control the reading procedure but it is not always possible to know in advance how many items are to be read. A way of getting round this is to use a special end-of-file marker. On the Acorn, Dragon and Tandy, the marker, called EOF, is automatically written onto the tape or disk whenever the CLOSE command is encountered. On the Commodore the STATUS keyword (or ST) is used instead. The Spectrum, unfortunately, does not use an end-of-file marker (but you can put in your own dummy marker as shown later).

So the last program can be rewritten to keep on printing out the data until the EOF marker is reached or, on the Commodore,



until the variable ST=64.

Here is the complete program from Line 1ØØ onwards so you can compare it to the earlier version. SAVE the old program, change the lines to the ones below and then try writing and reading the data again. You can choose which version suits you best.

As usual, if you have a disk drive, change the second number after OPEN to an 8.

```
1ØØ OPEN 1,1,1,“FILE”
1Ø5 Q$=CHR$(34):C$=“,”
11Ø Z=Ø
12Ø Z=Z+1
13Ø PRINT#1,Q$A$(Z)Q$C$Q$B$(Z)Q$C$
   Q$T$(Z)Q$
14Ø IF A$(Z)<>“” AND Z<5Ø THEN 12Ø
15Ø CLOSE 1
16Ø PRINT “DATA SAVED”
17Ø END
2ØØ CLR:OPEN 1,1,Ø,“FILE”
21Ø Z=Ø:DIMA$(5Ø),B$(5Ø),T$(5Ø)
22Ø Z=Z+1
23Ø INPUT#1,A$(Z),B$(Z),T$(Z)
235 PRINT A$(Z),B$(Z),T$(Z)
24Ø IF ST<>64 THEN 22Ø
25Ø CLOSE 1
```

```
1ØØ C=OPENOUT“FILE”
11Ø N=Ø:REPEAT
12Ø N=N+1
13Ø PRINT#C,A$(N),B$(N),T$(N)
14Ø UNTIL A$(N)=“” OR N=5Ø
15Ø CLOSE#C
16Ø PRINT “DATA SAVED”
17Ø END
2ØØ CLEAR:C=OPENIN“FILE”
21Ø DIM A$(5Ø),B$(5Ø),T$(5Ø)
22Ø N=Ø:REPEAT
23Ø N=N+1:INPUT#C,A$(N),B$(N),T$(N)
235 PRINTA$(N),B$(N),T$(N)
24Ø UNTIL EOF#C
25Ø CLOSE#C
```

Delete Line 2ØØ and change these lines. For tape:

```
1ØØ OPEN “O”, #−1, “FILE”
11Ø N=Ø
12Ø N=N+1
13Ø PRINT#−1,A$(N),B$(N),T$(N)
14Ø IF A$(N)<>“” AND N<5Ø THEN 12Ø
15Ø CLOSE#−1
16Ø PRINT“DATA SAVED”
17Ø END
2ØØ CLEAR 2ØØØ; DIM A$(5Ø),B$(5Ø),
   T$(5Ø):OPEN“I”,#−1,“FILE”
23Ø N=N+1: INPUT#
   −1,A$(N),B$(N),T$(N)
235 PRINT A$(N);B$(N);T$(N)
24Ø IF EOF(−1)=Ø THEN 23Ø
25Ø CLOSE#−1
```

For disk:

```
1ØØ CREATE“FILE”
11Ø N=Ø
12Ø N=N+1
13Ø FWRITE“FILE”;A$(N);“,”;B$(N);“,”; T$(N)
14Ø IF A$(N)<>“” AND N<5Ø THEN 12Ø
15Ø CLOSE
16Ø PRINT “DATA SAVED”
17Ø END
2ØØ CLEAR 2ØØØ: DIM A$(5Ø),B$(5Ø),
   T$(5Ø)
23Ø N=N+1:FREAD“FILE”;A$(N),B$(N),
   T$(N)
235 PRINTA$(N);B$(N);T$(N)
24Ø IF EOF(“FILE”)=Ø THEN23Ø
25Ø CLOSE
```

Line 24Ø in all these programs checks for the end of file and you can see the exact form to use for your computer. On the Commodore Line 1ØØ can also be written differently if you use a tape. Try this:

```
1ØØ OPEN 1,1,2,“FILE”
```

The number 2 in the OPEN command means ‘write an end-of-file marker when the file is closed’, and causes the tape to stop running when you come to the end of the file.

DUMMY TERMINATORS

An alternative way of checking for the end of the file is to have some kind of dummy variable. In fact this is the only way to check for the end of a Spectrum Microdrive file—it is not possible at all with a Spectrum tape file. The procedure is quite simple. In the original input routine, when data is first entered into the computer, a dummy value or word can be entered after the last proper item. This dummy could be the word END, or ZZZ or −999 or anything you would not normally expect to find in the real file. Alter these lines to see how the dummy value is used in practice. This time when you have finished entering the data type END instead of the next FIRST NAME, then RETURN or ENTER for the other two prompts.

This is only possible with the alternative Microdrive version that saves data one item at a time. Line 7Ø needs to be changed as shown, the writing and reading routines are given in full:

```
7Ø IF A$(N)<>“END” AND N<5Ø THEN
   GOTO 3Ø
1ØØ INPUT “ENTER DRIVE NUMBER ”;DRV
11Ø OPEN#4;“M”;DRV;“FILE”
12Ø L=Ø
125 L=L+1
13Ø PRINT#4;A$(L)'B$(L)'T$(L)
14Ø IF A$(L)<>“END” AND L<5Ø THEN
   GOTO 125
15Ø CLOSE#4
16Ø PRINT “DATA SAVED”
17Ø STOP
2ØØ CLEAR: DIM A$(5Ø,15): DIM B$(5Ø,15):
   DIM T$(5Ø,12)
2Ø5 INPUT “ENTER DRIVE NUMBER ”;DRV
21Ø OPEN#4;“M”;DRV;“FILE”
215 L=Ø
22Ø L=L+1
23Ø INPUT#4;A$(L);B$(L);T$(L)
235 PRINT A$(L);B$(L);T$(L)
24Ø IF A$(L)<>“END” AND L<5Ø THEN
   GOTO 22Ø
25Ø CLOSE#4
```

```
7Ø IF A$(N)<>“END” AND N<5Ø THEN 3Ø
14Ø IF A$(Z)<>“END” AND Z<5Ø THEN 12Ø
24Ø IF A$(Z)<>“END” AND Z<5Ø THEN
   22Ø
```

Make these changes to the EOF program:

```
7Ø UNTIL A$(N)=“END” OR N=5Ø
14Ø UNTIL A$(N)=“END” OR N=5Ø
24Ø UNTIL A$(N)=“END” OR N=5Ø
```

Make these changes to either the tape or disk version of the EOF program:

```
7Ø IF A$(N)<>“END” AND N<5Ø THEN 3Ø
14Ø IF A$(N)<>“END” AND N<5Ø THEN 12Ø
24Ø IF A$(N)<>“END” AND N<5Ø THEN
   23Ø
```

USING THE DATA

The file is simply a means of storing all of the data separately from the program. It is there to be read by whatever program will make the best use of it. You can, for example, use any of the sort routines given on pages 392 to 397 and 708 to 711 to sort the list into alphabetical order. Once sorted you can quickly search for specific items using a binary search program similar to the one on pages 926 and 927. By the time you have added on other routines to add and delete items you'll have a complete data file similar to the program starting on page 46. However all these routines are concerned with handling arrays rather than files, and the techniques involved have been covered elsewhere in *INPUT*.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT.*

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 44...

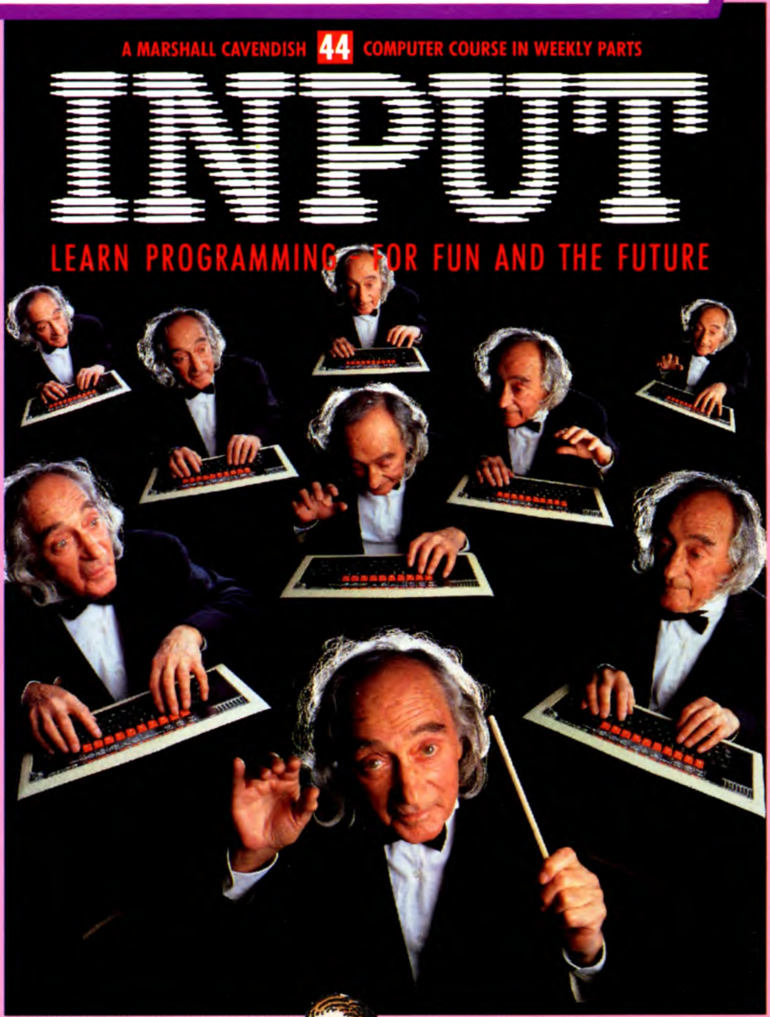
- ❏ *If you have favourite screen pictures or charts you want to keep, you can take a print-out using a SCREEN DUMP PROGRAM. There are two versions, a simple BASIC program, and a machine code routine for different tones*

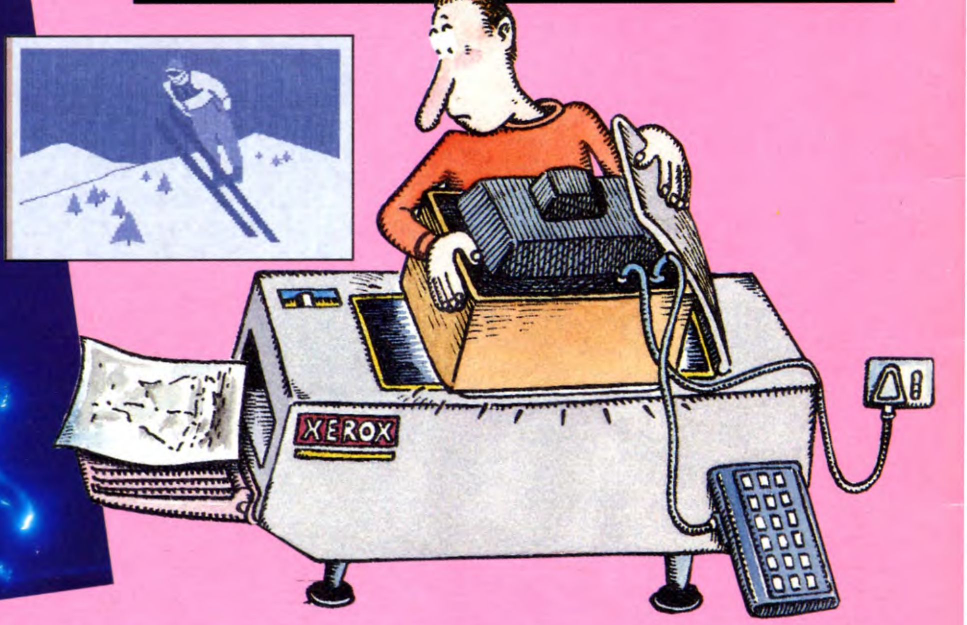
- ❏ *LANGUAGES looks at PASCAL, the structures language that many see as the most viable alternative to BASIC*

- ❏ *If you found the computer WARGAME easy to beat, see if you can still win after you have added MACHINE INTELLIGENCE*

- ❏ *The MUSIC COMPOSER continues by adding more of the listing that lets you compose, edit and play back*

- ❏ *The next routine to add in CLIFFHANGER lets your man JUMP to avoid the hazards that face him*


A MARSHALL CAVENDISH 44 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE



ASK YOUR NEWSAGENT FOR INPUT